1977-03

MICRO-COBOL: an implementation of Navy standard HYPO-COBOL for a microprocessor-based computer system

Craig, Alan Scott

Monterey, California. Naval Postgraduate School

P86 - CLOSE' Format should be CLOSE file-name.
P96 - OPEN " " " OPEN { }.

DI = Duplicate ID

MO = Memory Overflow.

SCR Routine screwed up decode calls for 9, Routine [illegible] 4 bytes.

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

MICRO-COBOL
AN IMPLEMENTATION OF
NAVY STANDARD HYPO-COBOL
FOR A MICROPROCESSOR-BASED COMPUTER SYSTEM

by

Alan Scott Craig

March 1977

Thesis Advisor: Gary A. Kildall

Approved for public release; distribution unlimited.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MICRO-COBOL an implementation of Navy Standard Hypo-Cobol for a microprocessor-based computer system		5. TYPE OF REPORT & PERIOD COVERED Masters Thesis; March 1977
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Alan Scott Craig		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE March 1977
		13. NUMBER OF PAGES 170
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

COBOL, compiler, formal grammar, microprocessor, microcomputer, LALR(1), HYPO-COBOL

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A compiler for ADPESO standard HYPO-COBOL has been implemented on a microcomputer. The implementation provides nucleus level constructs and file options from the ANSII COBOL package along with the PERFORM UNTIL construct from a higher level to give increased structural control. The language was implemented through a self-hosted compiler and run-time package

on an 8080 microcomputer-based system. Both compiler and interpreter can be executed in 12K bytes of user storage.

MICRO-COBOL
an implementation of Navy Standard Hypo-Cobol for a microprocessor-based computer system

by

Alan Scott Craig
Captain, United States Marine Corps
B.S., Brigham Young University, May 1971

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1977

ABSTRACT

A compiler for ADPESO standard HYPO-COBOL has been implemented on a microcomputer. The implementation provides nucleus level constructs and file options from the ANSII COBOL package along with the PERFORM UNTIL construct from a higher level to give increased structural control. The language was implemented through a self-hosted compiler and run-time package on an 8080 microcomputer-based system. Both compiler and interpreter can be executed in 12K bytes of user storage.

CONTENTS

I. INTRODUCTION

A. HISTORY OF COBOL

As indicated in the name, COBOL - COmmon Business Oriented Language - was intended to be a common standard computer programming language with consistent implementations on various machines. Backed heavily by the Department of Defense, COBOL has become a widely accepted language for data processing applications. Over the fifteen years of its existance the language has undergone several revisions and still continues to be upgraded and changed [1].

The evolution of COBOL has resulted in a large language containing numerous capabilities, many of which are not appropriate for a given machine nor desired by a class of users. For this reason the COBOL language is broken down into modules which may be implemented at various levels. The minimal standard COBOL, as currently defined, contains only the lowest levels of three modules out of the possible twelve modules which currently exist.

B. MOTIVATIONS OF HYPO-COBOL

None of the existing standard sets of COBOL modules fit the requirements of the Department of the Navy, and thus HYPO-COBOL was developed. Rather than taking one of the implementation levels described in the standard, another subset of the complete instruction set was developed which includes only parts of modules. HYPO-COBOL was designed to impose minimal requirements on a system for compiler support. Where possible, short constructs were used in the place of longer ones. Where multiple reserved words serve the same function in COBOL, the shortest form was used. There is no optional verbage in the language, and there are no duplicate constructs performing the same function.

Limits were placed on all statements that have a variable input format so that all statements have a fixed maximum length. Where possible, such constructs were removed completely from the language. In addition, user defined names were limited to twelve characters to reduce symbol table storage requirements.

Rather than include the standard levels of implementation for all of the modules, constructs were included only as required. In addition to low level constructs, the PERFORM UNTIL construct was included to allow better program structure. Further justification for the manner of subsetting and a highly detailed description of each element of the language is contained in the HYPO-COBOL Manual [10].

C. MICROCOMPUTERS

Current technological advances in the design of integrated computer components have lead to the proliferation of single chip central processors known as microcomputers. The number of chips produced and the varying capabilities of each product make generalizations very difficult. The term microcomputer, however, is generally used to describe a system built around one of these processors. Such a system would have memory, input and output capabilities, and timing circuts as well as a central processor. One chip systems with all of these capabilities are currently becoming available.

1. Hardware

The most significant factor in the proliferation of microcomputer-based systems has been their cost. Reasonably powerful central processors can currently be purchased for less than twenty dollars, resulting in the appearance of many new applications. Along with the low cost of the central processor have come low cost peripheral devices that are well suited to the speeds and capabilities of the microcomputers. In the case of traditional users of computers, the low cost of microcomputer hardware has led to new uses and to distributed processor networks. Changes in the cost and capabilities of microcomputers have been dramatic over the last several years, with more and more capabilities being offered at lower prices.

2. Software

Software has lagged far behind the developments in hardware for microcomputers. Most of the currently available systems do not support high level languages at all, and where supported, the languages are often systems languages rather than applications oriented languages. One of the restrictions imposed by many high level languages has been the requirement for cross-compiling on a more powerful machine [7]. In addition, some of the resident compilers require large amounts of memory. Recent work on versions of BASIC however, has led to quality resident compilers for scientific type calculations [6].

To allow the use of microprocessor systems in many of the proposed applications, languages need to be developed that will run on microcomputers without placing unreasonable demands on their capabilities and size. If the developments in hardware continue at their present rate, software will almost certainly continue to lag behind. However, current compiler construction techniques do seem to make it possible to provide the required languages, at least on the current types of hardware [3].

D. OBJECTIVES OF MICRO-COBOL

The major objective of this project was to implement HYPO-COBOL on an 8080 microcomputer-based system. As steps toward that objective, the following underlying goals were established: first, define HYPO-COBOL as an LALR(1) grammar [12]. Second, construct a compiler based on a table-driven parser for that LALR(1) grammar. Third, implement an interpreter to run the intermediate language instructions produced by the compiler.

While it was recognized that there would be difficulties in displaying the complete capabilities of the HYPO-COBOL language on the equipment currently available at the Naval Postgraduate School, it was considered feasible to implement a major portion of the subset with the current equipment and software.

One of the justifications for this project was the current standard policy of the Department of Defense to require all computers used in non-tactical environments to be capable of executing COBOL. In the case of the Department of the Navy, the standard that would need to be met for a microcomputer-based system is HYPO-COBOL.

Finally, it should be noted that there was no attempt to add to the HYPO-COBOL definition. One area of investigation was to test the feasibility of the subset. In defining the grammar, areas were found where additions could have been made, and future users may require enhanced capabilities to

make the language fit their requirements. Indications have been made, in the following sections, of places where changes seemed appropriate.

II. MICRO-COBOL MACHINE

A. GENERAL DESCRIPTION

The following sections describe the MICRO-COBOL pseudo-machine architecture in terms of allocated memory areas and pseudo-machine operations. The pseudo machine was the target machine for the compiler and was implemented through a programmed interpretation. The MICRO-COBOL machine has been given first, since all other system components can be described in terms of the target machine.

There were several ways to design the pseudo machine. The parser used produces operations in the order convenient for a stack machine, and other applications have used a simulation of a stack machine to interpret the output of the compiler [6]. The operations required for HYPO-COBOL did not require the use of a stack but could be designed as relatively independent operations. It would be possible to produce an interpreter that consisted of a set of subroutines which would be called directly by machine level operations on the 8080. The emitted code would then consist of instructions to load parameters and calls to the subroutines. This second idea was rejected due to the limited time available for the production of the project and because the code generation would then be very closely tied to the exact implementation of the interpreter. It was de-

cided to produce output code for a pseudo machine that would be defined to have all of the needed operations as basic instructions. The machine operators chosen contain all of the information required to perform one complete action required by the language.

The machine contains multiple parameter operators and a program counter that addresses the next instruction to be executed. Three registers are provided which hold eighteen digit numbers used for arithmetic operations along with a subscript stack that is used to compute subscript locations along with a set of flags that are used to pass branching information from one instruction to another.

Addresses in the machine are represented by 16 bit values. Any memory address greater than 20 hexadecimal is valid. Addresses less than 20 hexadecimal will be interpreted as having special significance. For example, addresses one through eight are reserved for subscript stack references. All other addresses in the machine are absolute addresses.

The arithmetic registers allow for the manipulation of signed numbers of up to eighteen decimal digits in length. Included in their representation is a sign indicator and the position of the assumed decimal point for the currently loaded number. While the form of the representation is not specified in the HYPO-COBOL document, it is necessary that there be no loss of precision for operations on numbers hav-

ing a full eighteen digits of significance.

There are two major types of numbers defined in the machine. The first is numbers in the DISPLAY mode. These numbers are represented in memory in the standard information exchange code for the peripherals. For microcomputers, the common representation would be in ASCII characters. These numbers may have separate signs indicated by "+" and "-" or may have a "zone" indicator added, denoting a negative sign. Packed decimal format is also available with numbers carried as sequential digit pairs stored in memory. The sign is indicated in the right-most position.

The following flags exist in the machine and can be checked by the instructions for a true or false value: BRANCH flag -- indicates if a branch is to be taken; END OF RECORD flag -- indicates that an end of input condition has been reached when an attempt was made to read input; OVERFLOW flag -- indicates the loss of information from a register due to a number exceeding the available size; INVALID flag -- indicates an invalid action in writing to a direct access storage device.

The following resources are required for a minimal implementation of this machine: a system input device capable of receiving low volume input, a system output device capable of displaying low volume output, and a direct access storage device capable of storing, reading, and writing files and programs.

B. MEMORY ORGANIZATION

Memory is divided into three major sections: (1) the data areas defined by the DATA DIVISION statements, (2) the code area, (3) and the constants area. No particular order of these sections is required. The first two areas assume the ability to both read and write, but the third only requires the ability to be read.

The data area contains variables defined by the DATA DIVISION statements, constants set in the WORKING STORAGE SECTION, and all file control blocks and buffers. These elements will be manipulated by the machine in accordance with the code instructions.

C. MACHINE OPERATIONS

1. Format

All of the machine operations consist of an operation number followed by a list of parameters. The sections that follow describe the various instructions, list the required parameters, and describe the actions taken by the machine in executing each instruction. As each instruction is fetched from memory, the program counter automatically increments by one.

2. Arithmetic operations

There are five arithmetic instructions which act only on the registers. In all cases, the result is placed

in register two. Operations are allowed to destroy the input values during the process of creating a result. Therefore, a number loaded into a register will not be available for a subsequent operation.

ADD: (addition). Sum the contents of register zero and register one.

Parameters: no parameters are required.

SUB: (subtract). Subtract register one from register zero.

Parameters: no parameters are required.

MUL: (multiply). Multiply register zero by register one.

Parameters: no parameters are required.

DIV: (divide). Divide register zero by the value in register one. The remainder is not retained.

Parameters: no parameters are required

RND: (round). Round register two to the last significant decimal place.

Parameters: no parameters are required.

3. Branching

All of the branching instructions are accomplished by changing the value of the program counter. Some are absolute branches and some test for condition flags that are set by the other instructions. Branches may also test the

state of the registers or perform direct comparisons on memory fields.

Several instructions use the same conditional branching conventions. First, the branch flag is checked for its current setting. If it is true, then a branch is made by changing the program counter to the value of the <branch address>. The branch flag is then set to false. If the flag was originally false, the program counter is incremented to the next sequential instruction.

BRN: (branch to an address). Load the program counter with the <branch address>.
Parameters: <branch address>

The next three instructions share a common format. The memory field addressed by the <memory address> is checked for the <address length>, and if all the characters match the test condition, then the branch flag is complemented. A conditional branch is taken after the test.
Parameters: <memory address> <address length> <branch address>

CAL: (compare alphabetic). Compare a memory field for alphabetic characters.

CNS: (compare numeric signed). Compare a field for numeric characters allowing for a sign character.

CNU: (compare numeric unsigned). Compare a field for numeric characters only.

DEC: (decrement a count and branch if zero). Decrement the value of the <address counter> by one, and if the result is zero, the program counter is set to the address given. If the result is not zero, then the program counter is incremented by four. If the result is zero before decrementing, the branch is taken.
Parameters: <address counter> <branch address>

EOR: (branch on end of records flag). If the end-of-records flag is true, it is set to false and the program counter is set to the <branch address>. If false, the program counter is incremented by two.
Parameters: <branch adress>

GDP: (go to - depending on). The memory location addressed by the <number adress> is read for the number of bytes indicated by the <memory length>. This number indicates which of the <branch addresses> is to be used. The first parameter is a bound on the number of branch addresses. If the number is within the range, the program counter is set to the indicated address. An out of bounds value causes the program counter to be advanced to the next sequential instruction.
Parameters: <bound number - byte> <memory length> <memory address> <branch addr-1> <branch addr-2> ... <branch addr-n>

INV: (branch if invalid-file-action flag true). If the invalid-file-action flag is true, then it is set to false, and the program counter is set to the branch ad-

dress. If it is false, the program counter is incremented by two.

Parameters: <branch address>

PER: (perform). The code address pointed to by the <change address> is loaded with the value of the <return address>. The program counter is then set to the <branch address>.

Parameters: <branch address> <change address> <return address>

RET: (return). If the value of the <branch address> is not zero, then the program counter is set to its value, and the <branch address> is set to zero. If the <branch address> is zero, the program counter is incremented by two.

Parameters: <branch address>

REQ: (register equal). This instruction checks for a zero value in register two. If it is zero, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

RGT: (register greater than). Register two is checked for a negative sign. If present, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

RLT: (register less than). Register two is checked for a positive sign, and if present, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

SER: (branch on size error). If the overflow flag is true, then the program counter is set to the branch address, and the overflow flag is set to false. If it is false, then the program counter is incremented by two.

Parameters: <branch address>

The next three instructions all perform the same function and have the same general format. They compare two strings and perform a conditional branch. If the test condition is true, the branch flag is complemented prior to taking the conditional branch.

Parameters: <string addr-1> <string addr-2> <length - address> <branch address>

SEQ: (strings equal). Compare two string for equal characters.

SGT: (string greater than). Compare string one for greater than string two.

SLT: (string less than). Compare string one for less than string two.

4. Moves

The machine supports a variety of move operations for various formats and types of data. It does not support direct moves of numeric data from one memory field to another. Instead, all of the numeric moves go through the regis-

ters. This greatly reduced the number of instructions since all of the numeric types need to be supported by moves into and out of the registers for arithmetic operations.

The next seven instructions all perform the same function. They load a register with a numeric value and differ only in the type of number that they expect to see in memory at the <number address>. All seven cause the program counter to be incremented by five. Their common format is given below.

Parameters: <number address> <byte length> <byte decimal count> <byte register to load>

LDD: (load a numeric literal). Note that the decimal point indicator is not set in this instruction format. The literal will have an actual decimal point in it if required.

LD1: (load a numeric field).

LD2: (load a numeric field with an internal trailing sign).

LD3: (load a numeric field with an internal leading sign).

LD4: (load a numeric field with a separate leading sign).

LD5: (load a numeric field with a separate trailing sign).

LD6: (load a packed numeric field).

MED: (move into a alphanumeric edited field). The edit mask is loaded into the <to address> to set up the move, and then the <from address> information is loaded. The program counter is incremented by ten.

Parameters: <to address> <from address> <length of move> <edit mask address> <edit mask length>

MNE: (move into a numeric edited field). First the edit mask is loaded into the receiving field, and then the information is loaded. Any decimal point alignment required will be performed. If truncation of significant digits is a side effect, the overflow flag is not set. The program counter is incremented by twelve.

Parameters: <to address> <from address> <address length of move> <edit mask address> <address mask length> <byte to decimal count> <byte from decimal count>

MOV: (move into an alphanumeric field). The memory field given by the <to address> is filled by the from field for the <move length> and then filled with blanks in the following positions for the <fill count>.

Parameters: <to address> <from address> <address move length> <address fill count>

SI2: (store immediate register two). The contents of register two are stored into register zero and the decimal count and sign are indicators set.

Parameters: none.

The store instructions are grouped in the same order as the load instructions. Register two is stored into memory at the indicated location. Any alignment is performed, and if a non-zero leading digit is truncated by the operation, the overflow flag is set. All five of the store instructions cause the program counter to be incremented by four. The format for these instructions is as follows.

Parameters: <address to store into> <byte length> <byte decimal count>

ST0: (store into a numeric field).

ST1: (store into a numeric field with an internal trailing sign).

ST2: (store into a numeric field with an internal leading sign).

ST3: (store into a numeric field with a separate trailing sign).

ST4: (store into a numeric field with a separate leading sign).

ST5: (store into a packed numeric field).

5. Input-output

The following instructions perform input and output operations. The required operations are specified in the HYPO-COBOL manual, but the exact definitions of file formats and access methods are not defined. Files in this machine

are defined as having the following characteristics: they are either sequential or random, and, in general, files created in one mode are not required to be readable in the other mode. Standard files consist of fixed length records, and variable length files need not be readable in a random mode. Further, there must be some character or charcter string that delimits a variable length record.

ACC: (accept). Read from the system input device into memory at the location given by the <memory address>. The program counter is incremented by three.

Parameters: <memory address> <byte length of read>

CLS: (close). Close the file whose file control block is addressed by the <fcb address>. The program counter is incremented by two.

Parameters: <fcb address>

DIS: (display). Print the contents of the data field pointed to by <memory address> on the system output device for the indicated length. The program counter is increment-ed by three.

Parameters: <memory address> <byte length>

There are three open instructions with the same for-mat. In each case, the file defined by the file control block referenced will be opened for the mode indicated. The program counter is incremented by two.

Parameters: <fcb address>

OPN: (open a file for input).

OP1: (open a file for output).

OP2: (open a file for both input and output). This is only valid for files on a random access device.

The following file actions all share the same format. Each performs a file action on the file referenced by the file control block. The record to be acted upon is given by the <record address>. The program counter is incremented by six.

Parameters: <fcb address> <record address> <record length address>

DLS: (delete a record from a sequential file). Remove the record that was just read from the file. The file is required to be open in the input-output mode.

RDF: (read a sequential file). Read the next record into the memory area.

WIF: (write a record to a sequential file). Append a new record to the file.

RVL: (read a variable length record).

WVL: (write a variable length record).

RWS: (rewrite sequential). The rewrite operation writes a record from memory to the file, overlaying the last record that was read from the device. The file must be open

in the input-output mode.

The following file actions require random files rather than sequential files. They all make use of a random file pointer which consists of a <relative address> and a <relative length>. The memory field holds the number to be used in disk operations or contains the relative record number of the last disk action. The relative record number is the record count on the file starting with one. After the file action, the program counter is incremented by nine.

Parameters: <fcb address> <record address> <record length - address> <relative address> <relative length - byte>.

DLR: (delete a random record). Delete the record addressed by the relative record number.

RRR: (read random relative). Read a random record relative to the record number.

RRS: (read random sequential). Read the next sequential record from a random file. The relative record number of the record read is loaded into the memory reference.

RWR: (rewrite a random record).

WRR: (write random relative). Write a record into the area indicated by the memory reference.

WRS: (write random sequential). Write the next sequential record to a random file. The relative record

number is returned.

6. Special instructions

The remaining instructions perform special functions required by the machine that do not relate to any of the previous groups.

NOT: (negitive test). Negate the value of the branch flag.

Parameters: no parameters are required.

LDI: (load a code address direct). Load the <code address> with the number indicated by the <memory address>.

Parameters: <code address> <memory address> <length - byte>

SCR: (calculate a subscript). Load the subscript stack with the value indicated from memory. The address loaded into the stack is the <initial address> plus an offset. Multiplying the <field length> by the number in the <memory reference> gives the offset value.

Parameters: <initial address> <field length> <memory reference> <memory length> <stack level>

STD: (stop with display). Display the indicated information and then stop.

Parameters: <memory address> <length - byte>

STP: (stop). terminate the actions of the machine.

Parameters: no parameters are required.

The following instructions are used in setting up the machine environment and cannot be used in the normal execution of the machine.

BST: (backstuff). Resolve a reference to a label. Labels may be referenced prior to their definition, requiring a chain of resolution addresses to be maintained in the code. The latest location to be resolved is maintained in the symbol table and a pointer at that location indicates the next previous change. A zero pointer indicates no prior occurrences of the label. The code address referenced by <change address> is examined and if it contains zero, it is loaded with the <new address>. If it is not zero, then the contents are saved, and the process is repeated with the saved value as the change address after loading the <new address>.

Parameters: <change address> <new address>

INT: (initialize memory). Load memory with the <input string> for the given length at the <memory address>.

Parameters: <memory address> <address length> <input string>

SCD: (start code). Set the initial value of the program counter.

Parameters: <start address>

TER: (terminate). Terminate the initialization process and start executing code.

Parameters: no parameters are required.

III. MICRO-COBOL IMPLEMENTATION

A. COMPILER IMPLEMENTATION

1. General method

The LALR parser-table construction programs used here are based on the work of Knuth [9]. His work defines two methods of testing a grammar to see if it is LR(k). One of these methods leads to the creation of a set of tables that can be used to drive the parse actions of a compiler. While difficult to implement in the form given by Knuth, the method has been developed in usable form for subsets of the grammars that are LR(k). References 2 and 3 contain detailed discussions of the methods currently available. The algorithm used to develop the tables for the MICRO-COBOL compiler was developed by W. Lalonde [12].

The compiler was designed to read the source language statements from a diskette or other mass storage device, extract the needed information for the symbol table, and write the output code back onto the diskette all in one pass of the source program. The grammar was initially defined for the entire language, but the size constraints placed on the implementation required smaller tables. The grammar was then defined in two parts which run in succession. The major method of passing information from the

first part to the second is by placing the information in the symbol table.

The output code from the compiler consists of the operations that have been previously defined. They were designed as an intermediate language that would be executed by the interpreter described in section B. The vast differences between the operations available for the target computer and the operations necessary to support COBOL made this approach easier than 8080 machine code.

2. Control flow

The compiler has been designed so that the operation of the two parts would be transparent to the user. When the first part is loaded it brings in with its code a reader program which loads the second file automatically. Prior to calling the reader program, the first part writes any pending code to the disk and loads all toggles to a common area ready to be read by the second part.

Internally, the control of the two parts is identical. The parser is called after initialization and runs until it either finishes its task or reaches an unrecoverable error state. The major subroutines in the compiler are the scanner and the production case statement. Both are controlled in their actions by the parser.

3. Internal structures

The major internal structure is the symbol table. It was designed as a list where the elements in the list are the descriptions of the various symbols in the program. As new symbols are encountered they are added to the end of the list. Symbols already in the list can be accessed through the use of a "current symbol pointer." The location of items in the list is determined by checking the identifier against a hash table that points to the first entry in the symbol table with that hash code. A chain of collision addresses is maintained in the symbol table which links entries which have the same hash value.

All of the items in the symbol table contain the following information: a collision field, a type field, the length of the identifier, and the address of the item. If an item in the symbol table is a data field, the following information is included in the table: the length of the item, the level of the data field, an optional decimal count, an optional multiple occurrence count, and the address of the edit field, if required. If the item is a file name then the following additional information is included: the file record length, the file control block address, and the optional symbol table location of the relative record pointer. If the item is a label, then the only additional information is the location of the return instruction at the end of the paragraph or section.

In addition to the symbol table, two stacks are used for storing information: the level stack and the identifier stack. In both cases, they are used to hold pointers to entries in the symbol table. The identifier stack is used to collect multiple occurrences in such statements as the GO TO - DEPENDING statement. The level stack is used to hold information about the various levels that make up a record description.

The parser has control of a set of stacks that are used in the manipulation of the parse states. In addition to the state stack that is required by the parser, part one has a value stack and part two has two different value stacks that operate in parallel with the parser state stack. The use of these stacks is described below.

4. Part one

The first part of the compiler is primarily concerned with building the symbol table that will be used by the second part. The actions corresponding to each parse step are explained in the sections that follow. In each case, the grammar rule that is being applied is given, and an explanation of what program actions take place for that step has been included. In describing the actions taken for each parse step there has been no attempt to describe how the symbol table is constructed or how the values are preserved on the stack. The intent of this section is to describe what information needs to be retained and at what

point in the parse it can be determined. Where no action is required for a given statement, or where the only action is to save the contents of the top of the stack, no explaination is given. Questions regarding the actual manipulation of information should be resolved by consulting the programs.

1 <program> ::= <id-div> <e-div> <d-div> PROCEDURE

Reading the word PROCEDURE terminates the first part of the compiler.

2 <id-div> ::= IDENTIFICATION DIVISION. PROGRAM-ID.
<comment> . <auth> <date> <sec>

3 <auth> ::= AUTHOR . <comment> .

4 ¦ <empty>

5 <date> ::= DATE-WRITTEN . <comment> .

6 ¦ <empty>

7 <sec> ::= SECURITY . <comment> .

8 ¦ <empty>

9 <comment> ::= <input>

10 ¦ <comment> <input>

11 <e-div> ::= ENVIRONMENT DIVISION . CONFIGURATION SECTION.
<scr-obj> <i-o>

12 <src-obj> ::= SOURCE-COMPUTER . <comment> <debug> .
OBJECT-COMPUTER . <comment> .

13 <debug> ::= DEBUGGING MODE

Set a scanner toggle so that debug lines will be read.

14 ¦ <empty>

15 <i-o> ::= INPUT-OUTPUT SECTION . FILE-CONTROL .

<file-control-list> <ic>

16 | <empty>

17 <file-control-list> ::= <file-control-entry>

18 | <file-control-list> <file-control-entry

19 <file-control-entry> ::= SELECT <id> <attribute-list> .

At this point all of the information about the file has been collected and the type of the file can be determined. File attributes are checked for compatability and entered in the symbol table.

20 <attribute-list> ::= <one attrib>

21 | <attribute-list> <one attrib>

22 <one-attrib> ::= ORGANIZATION <org-type>

23 | ACCESS <acc-type> <relative>

24 | ASSIGN <input>

A file conrol block is built for the file using an INT operator.

25 <org-type> ::= SEQUENTIAL

No information needs to be stored since the default file organization is sequential.

26 | RELATIVE

The relative attribute is saved for production 19.

27 <acc-type> ::= SEQUENTIAL

This is the default.

28 | RANDOM

The random access mode needs to be saved for production 19.

29 <relative> ::= RELATIVE <id>

The pointer to the identifier will be retained by the

current symbol pointer, so this production only saves a flag on the stack indicating that the production did occur.

30 | <empty>

31 <ic> ::= I-O-CONTROL . <same-list>

32 | <empty>

33 <same-list> ::= <same-element>

34 | <same-list> <same-element>

35 <same-element> ::= SAME <id-string> .

36 <id-string> ::= <id>

37 | <id-string> <id>

38 <d-div> ::= DATA DIVISION . <file-section> <work> <link>

39 <file-section> ::= FILE SECTION . <file-list>

Actions will differ in production 64 depending upon whether this production has been completed. A flag needs to be set to indicate completion of the file section.

40 | <empty>

The flag, indicated in production 39, is set.

41 <file-list> ::= <file-element>

42 | <file-list> <file-element>

43 <files> ::= FD <id> <file-control> . <record-description>

This statement indicates the end of a record description, and the length of the record and its address can now be loaded into the symbol table for the file name.

44 <file-control> ::= <file-list>

45 | <empty>

```
46  <file-list> ::= <file-element>
47                | <file-list> <file-element>
48  <file-element> ::= BLOCK <integer> RECORDS
49                   | RECORD <rec-count>
```

The record length can be saved for comparison with the calculated length from the picture clauses.

```
50                   | LABEL RECORDS STANDARD
51                   | LABEL RECORDS OMITTED
52                   | VALUE OF <id-string>
53  <rec-count> ::= <integer>
54                | <integer> TO <integer>
```

The TO option is the only indication that the file will be variable length. The maximum length must be saved.

```
55  <work> ::= WORKING-STORAGE SECTION . <record-description>
56           | <empty>
57  <link> ::= LINKAGE SECTION . <record-description>
58           | <empty>
59  <record-description> ::= <level-entry>
60                         | <record-description> <level-entry>
61  <level-entry> ::= <integer> <data-id> <redefines>
                      <data-type> .
```

The level entry needs to be loaded into the level stack. The level stack is used to keep track of the nesting of field definitions in a record. At this time there may be no information about the length of the item being defined, and its attributes may depend entirely upon its constituent fields. If there is a

pending literal, the stack level to which it applies is saved.

62 <data-id> ::= <id>

63 | FILLER

An entry is built in the symbol table to record information about this record field. It cannot be used explicitly in a program because it has no name, but its attributes will need to be stored as part of the total record.

64 <redefines> ::= REDEFINES <id>

The redefines option gives new attributes to a previously defined record area. The symbol table pointer to the area being redefined is saved so that information can be transfered from one entry to the other. In addition to the information saved relative to the redefinition, it is necessary to check to see if the current level number is less than or equal to the level recorded on the top of the level stack. If this is true, then all information for the item on the top of the stack has been saved and the stack can be reduced.

65 | <empty>

As in production 64, the stack is checked to see if the current level number indicates a reduction of the level stack. In addition, special action needs to be taken if the new level is 01. If an 01 level is encountered at this production prior to production 39 or 40 (the end of the file area), it is an implied rede-

finition of the previous 01 level. In the working storage section, it indicates the start of a new record.

66 <data-type> ::= <prop-list>

67 | <empty>

68 <prop-list> ::= <data-element>

69 | <prop-list> <data-element>

70 <data-element> ::= PIC <input>

The <input> at this point is the character string that defines the record field. It is analyzed and the extracted information is stored in the symbol table.

71 | USAGE COMP

The field is defined to be a packed numeric field.

72 | USAGE DISPLAY

The DISPLAY format is the default, and thus no special action occurs.

73 | SIGN LEADING <separate>

This production indicates the presence of a sign in a numeric field. The sign will be in a leading position. If the <separate> indicator is true, then the length will be one longer than the picture clause, and the type will be changed.

74 | SIGN TRAILING <separate>

The same information required by production 73 must be recorded, but in this case the sign is trailing rather than leading.

75 | OCCURS <integer>

The type must be set to indicate multiple occurrences,

and the number of occurrences saved for computing the space defined by this field.

76 ¦ SYNC <direction>

Syncronization with a natural boundary is not required by this machine.

77 ¦ VALUE <literal>

The field being defined will be assigned an initial value determined by the value of the literal through the use of an INT operator. This is only valid in the WORKING-STORAGE SECTION.

78 <direction> ::= LEFT

79 ¦ RIGHT

80 ¦ <empty>

81 <separate> ::= SEPARATE

The separate sign indicator is set on.

82 ¦ <empty>

83 <literal> ::= <input>

The input string is checked to see if it is a valid numeric literal, and if valid, it is stored to be used in a value assignment.

84 ¦ <lit>

This literal is a quoted string.

85 ¦ ZERO

As is the case of all literals, the fact that there is a pending literal needs to be saved. In this case and the three following cases, an indicator of which literal constant is being saved is all that is required. The literal value can be reconstructed

later.

86 | SPACE

87 | QUOTE

88 <integer> ::= <input>

The input string is converted to an integer value for later internal use.

89 <id> ::= <input>

The input string is the name of an identifier and is checked aginst the symbol table. If it is in the symbol table, then a pointer to the entry is saved. If it is not in the symbol table, then an entry is added and the address of that entry is saved.

5. Part two

The second part includes all of the PROCEDURE DIVISION, and is the part where code generation takes place. As in the case of the first part, there was no intent to show how various pieces of information were retrieved but only what information was used in producing the output code.

1 <p-div> ::= PROCEDURE DIVISION <using> .
<proc-body> END .

This production indicates termination of the compilation. If the program has sections, then it will be necessary to terminate the last section with a RET 0 instruction. The code will be ended by the output of a TER operation.

2 <using> ::= USING <id-string>

3 | <empty>

4 <id-string> ::= <id>

The identifier stack is cleared and the symbol table address of the identifier is loaded into the first stack location.

5 | <id-string> <id>

The identifier stack is incremented and the symbol table pointer stacked.

6 <proc-body> ::= <paragraph>

7 | <proc-body> <paragraph>

8 <paragraph> ::= <id> . <sentence-list>

The starting and ending address of the paragraph are entered into the symbol table. A return is emitted as the last instruction in the paragraph (RET 0). When the label is resolved, it may be necessary to produce a BST operation to resolve previous references to the label.

9 | <id> SECTION .

The starting address for the section is saved. If it is not the first section, then the previous section ending address is loaded and a return (RET 0) is output. As in production 8, a BST may be produced.

10 <sentence-list> ::= <sentence>

11 | <sentence-list> <sentence> .

12 <sentence> ::= <imperative>

13 | <conditional>

14 | ENTER <id> <opt-id>

This construct is not implemented. An ENTER allows

statements from another language to inserted in the source code.

15 <imperative> ::= ACCEPT <subid>

ACC <address> <length>

16 | <arithmetic>

17 | CALL <lit> <using>

This is not implemented.

18 | CLOSE <id>

CLS <file control block address>

19 | <file-act>

20 | DISPLAY <lit/id> <opt-lit/id>

The display operator is produced for the first literal or identifier (DIS <address> <length>). If the second value exists, the same code is also produced for it.

21 | EXIT <program-id>

RET 0

22 | GO <id>

BRN <address>

23 | GO <id-string> DEPENDING <id>

GDP is output, followed by a number of parameters: <the number of entries in the identifier stack> <the length of the depending identifier> <the address of the depending identifier> <the address of each identifier in the stack>.

24 | MOVE <lit-id> TO <subid>

The types of the two fields determine the move that is generated. Numeric moves go through register two using a load and a store. Non-numeric moves depend upon

the result field and may be either MOV, MED or MNE. Since all of these instructions have long parameter lists, they have not been listed in detail.

25 | OPEN <type-action> <id>

This produces either OPN, OP1, or OP2 depending upon the <type-action>. Each of these is followed by a file control block address.

26 | PERFORM <id> <thru> <finish>

The PER operation is generated followed by the <branch address> <the address of the return statement to be set> and <the next instruction address>.

27 | <read-id>

28 | STOP <terminate>

If there is a terminate message, then SPD is produced followed by <message address> <message length>. Otherwise STP is emitted.

29 <conditional> ::= <arithmetic> <size-error> <imperative>

A BST operator is output to complete the branch around the imperative from production 65.

30 | <file-act> <invalid> <imperative>

A BST operator is output to complete the branch from production 64.

31 | IF <condition> <action> ELSE <imperative>

Two BST operators are required. The first fills in the branch to the ELSE action. The second completes the branch around the <imperative>.

32 | <read-id> <special> <imperative>

A BST is produced to complete the branch around the

<imperative>.

33 <Arithmetic> ::= ADD <l/id> <opt-l/id> TO <subid> <round>

The existence of multiple load and store instructions make it difficult to indicate exactly what code will be generated for any of the arithmetic instructions. The type of load and store will depend on the nature of the number involved, and in each case the standard parameters will be produced. This parse step will involve the following actions: first, a load will be emitted for the first number into regster zero. If there is a second number, then a load into register one will be produced for it, followed by an ADD and a SII. Next a load into register one will be generated for the result number. Then an ADD instruction will be emitted. Finally, if the round indicator is set, a RND operator will be produced prior to the store.

34 | DIVIDE <l/id> INTO <subid> <round>

The first number is loaded into register zero. The second operand is loaded into register one. A DIV operator is produced, followed by a RND operator prior to the store, if required.

35 | MULTIPLY <l/id> BY <subid> <round>

The multiply is the same as the divide except that a MUL is produced.

36 | SUBTRACT <l/id> <opt-l/id> FROM <subid> <round>

Subtaction generates the same code as the ADD except that a SUB is produced in place of the last ADD.

37 <file-act> ::= DELETE <id>

Either a DLS or a DLR will be produced along with the required parameters.

38 | REWRITE <id>

Either a RWS or a RWR is emitted, followed by parameters.

39 | WRITE <id> <special-act>

There are four possible write instructions: WTF, WVL, WRS, and WRR.

40 <condition> ::= <lit/id> <not> <cond-type>

One of the compare instructions is produced. They are CAL, CNS, CNU, RGT, RLT, REQ, SGT, SLT, and SEQ. Two load instructions and a SUB will also be emitted if one of the register comparisons is required.

41 <cond-type> ::= NUMERIC

42 | ALPHABETIC

43 | <compare> <lit/id>

44 <not> ::= NOT

NEG

45 | <empty>

46 <compare> ::= GREATER

47 | LESS

48 | EQUAL

49 <ROUND> ::= ROUNDED

50 | <empty>

51 <terminate> ::= <literal>

52 | RUN

53 <special> ::= <invalid>

54 | END

An ERO operator is produced followed by a zero. The zero acts as a filler in the code and will be back-stuffed with a branch address. In this production and several of the following, there is a forward branch on a false condition past an imperative action. For an example of the resolution, examine production 32.

55 <opt-id> ::= <subid>

56 | <empty>

57 <action> ::= <imperative>

BRN 0

58 | NEXT SENTENCE

BRN 0

59 <thru> ::= THRU <id>

60 | <empty>

61 <finish> ::= <l/id> TIMES

LDI <address> <length> DEC 0

62 | UNTIL <condition>

63 | <empty>

64 <invalid> ::= INVALID

INV 0

65 <size-error> ::= SIZE ERROR

SER 0

66 <special-act> ::= <when> ADVANCING <how-many>

67 | <empty>

68 <when> ::= BEFORE

69 | AFTER

70 <how-many>::= <integer>

71 ¦ PAGE

72 <type-action> ::= INPUT

73 ¦ OUTPUT

74 ¦ I-O

75 <subid> ::= <subscript>

76 ¦ <id>

77 <integer> ::= <input>

The value of the input string is saved as an internal number.

78 <id> ::= <input>

The identifier is checked aginst the symbol table, if it is not present, it is entered as an unresolved label.

79 <l/id> ::= <input>

The input value may be a numeric literal. If so, it is placed in the constant area with an INT operand. If it is not a numeric literal, then it must be an identifier, and it is located in the symbol table.

80 ¦ <subscript>

81 ¦ ZERO

82 <subscript> ::= <id> (<input>)

If the identifier was defined with a USING option, then the input string is checked to see if it is a number or an identifier. If it is an identifier, then an SCR operator is produced.

83 <opt-l/id> ::= <l/id>

84 ¦ <empty>

85 <nn-lit> ::= <lit>

The literal string is placed into the constant area using an INT operator.

86 | SPACE

87 | QUOTE

88 <literal> ::= <nn-lit>

89 | <input>

The input value must be a numeric literal to be valid and is loaded into the constant area using an INT.

90 | ZERO

91 <lit/id> ::= <l/id>

92 | <nn-lit>

93 <opt-lit/id> ::= <lit/id>

94 | <empty>

95 <program-id> ::= <id>

96 | <empty>

97 <read-id> ::= READ <id>

There are four read operations: RDF, RVL, RRS, and RRR.

The output code file is the only product of the compiler that is retained. All of the needed information has been extracted from the symbol table, and it is not required by the interpreter. Code will be generated for all programs including those that contain errors and can be examined through the use of the decode program. This program translates the output file into a listing of code operators followed by the parameters.

B. INTERPRETER IMPLEMENTATION

1. General structure

The format that has been presented for the output code determines the general form of the interpreter. If it had not been possible to transform the instructions from the compiler into a set of call-like commands, it would have been necessary to implement a stack in the interpreter. In general, the interpreter contains a large "case statement" which decodes each operation and either calls subroutines to perform the required actions or acts directly on the run-time environment to control the actions of the interpreter. All communication between instructions is done through common areas in the program where information can be stored for later use.

The design of the interpreter has been modularized in an attempt to allow easy transition to other hardware configurations and operating systems. If desired, any section of the instructions could be implemented in assembly language modules or could be passed to the operating system for action. The entire system has been coded in PL/M for consistency, ease of development, and maximum portability [7].

2. Code modules

The following sections explain the interpreter by noting the specific manner in which the machine instructions

defined in section II-C have been implemented. The divisions are the same as those in section II-C.

a. Arithmetic instructions

Since the machine was defined as having only one set of arithmetic registers, it was necessary to convert all numeric input to one form. The packed decimal format was chosen as the format that would be used in the registers. This conversion process slows down the arithmetic operations slightly, but the reduction of the interpreter memory size was considered more important.

All of the arithmetic operations take place in a set of three work areas or registers. Each of these areas is ten bytes long and can contain an eighteen digit number with one fill character on each end. The extra space facilitates checking for overflow and also makes rounding operations easier. The language does not support the COMPUTE verb, so no storage of intermediate results is required from one instruction to another.

All of the arithmetic instructions use the packed decimal feature of the 8080 as a basis for their actions. Each of the instructions depends on the basic operation of adding two registers: subtraction is accomplished using nines complement arithmetic, multiplication is done through a shift and add algorithm, and division by a shift and subtract method.

If the amount of computations required by a given application make it necessary to speed up these instructions, they could be replaced by a package in assembly language. Extending the grammar to include the COMPUTE verb would require changes in the compiler to allow for temporary locations, but it could be included.

b. Branching

The operation of the interpreter is controlled by a program counter that points to the next operation to be performed. All branching is done by changing the normal sequential order of execution of instructions. In addition to acting directly on the program counter, branching instructions use the branch flag to determine when changes should be made. All of the addresses that point to code are absolute addresses and can be loaded directly into the program counter.

c. Input-output operations

All of the input and output operations use the CP/M interface capabilities [5]. The program expects to see the files in the form that the CP/M editor would have created them. The physical records on the disk are assumed to be 128 bytes in length and have all logical records ending with a carriage-return and a line-feed sequence. There is only one type of file under CP/M, so all restrictions on mixing modes of files are removed for fixed length files. Files created in one program as sequential can be accessed as ran-

dom files in another program. Variable length files cannot be accessed in a random fashion because there is no way to compute the starting address of each record.

Where possible, the interface routines have been localized in the programs to simplify transportation to another operating environment. Items relating to file control blocks, disk record lengths, and other system parameters have been established as literals in the programs, rather than entered as numbers, so that changes will not have to be made throughout the code.

a. Moves

As noted previously, the machine lacks numeric moves. There were two major reasons for leaving out the various moves of numeric data. The first was that the added moves would have required more program space, and the second was to simplify the coding and checking of the program. Since all of the numeric types are supported with register load and store operations, any move can be accomplished by a load into register two and a store into the result field.

Alpha-numeric moves are supported as direct moves from memory to memory. If speed is required for a numeric move, the fields concerned can be redefined as alpha-numeric and the memory move used. However, this type of move will only work on two numbers that have exactly the same representation in the computer.

Edited moves also are from memory to memory, but they involve several additional steps. The edit mask is loaded into the result field before any characters are loaded, and each character in both the receiving field and the sending field is examined to determine what action should be taken in addition to a move.

3. Limitations

The MICRO-COBOL implementation did not lend itself to support of the Interprogram Communications Module. There was no capability in the operating system to dump the memory image onto the disk or to restore it. It would be possible to implement such a supervisor call, or a one way call could perhaps be implemented from one program to another without the posibility of a return to the calling program. If required by an application where modification of the operating system was not practical, a small overlay program could be written as an independent function to be loaded with the interpreter. If large systems are to be run on microcomputers with minimal memory, some type of interprogram communications would greatly facilitate their design.

C. SOFTWARE TOOLS

As in any software development, one of the things that was most important to the success of this project was the software support for the development effort. This system was developed on the 360/67 rather than on the 8080. Using

the Intel INTERP program [8] and the CP/M simulator developed by at the Naval Postgraduate School [11], it was possible to both compile programs on CP/CMS and run the generated code. This facility removed the necessity of transporting code from the 360 to the 8080 for testing and greatly improved the productivity.

Using the simulator did not result in exactly the same product as would have been developed if the project had been done entirely on the 8080. It was not possible to load a program on the simulator without destroying the core image currently in the simulator. In particular, the first part of the compiler could not leave the symbol table for the second part if the second part was loaded by a normal load. This problem was resolved by writing a set of small programs that read in the sequence of compiler components from simulated memory image files. These programs have been included in this document so that, if future work is done, the simulator could be used again.

IV. CONCLUSIONS

This project demonstrates the feasibility of applying modern compiler construction techniques to the implementation of a language developed prior to the work on formal grammars. Not only is it possible to construct a compiler for HYPO-COBOL using an LALR(1) parser, but the resulting programs are highly compact. This allows the implementation of the compiler on smaller machines and increases the number of target systems.

Only a limited number of programs have been written using the compiler, and no attempt has been made to train others in its use. However, adapting to the subset should not be a major problem for a programmer experienced in writing standard COBOL. There have been no extensive timing tests of the system, but current indications are that both the compiler and interpreter operate at an acceptable rate.

There are several areas that could be enhanced in this implementation of HYPO-COBOL. One of these areas is the interprogram communication module. Due to the limitations on core size usually imposed by microcomputer systems, it would be very helpful to be able to compile a set of programs that could be used together as a single module. Several ideas were presented in the body of this paper which indicate how the interprogram communication module could be developed.

The GIVING option for arithmetic statements could be added to the grammar. This option would improve comptational programs, and could be supported without change to the existing interpreter. As discussed previously, the COMPUTE verb could be added if desired, but it would require greater changes both to the grammar and to the interpreter.

Programmers that have used COBOL in a standard implementation will find the appearance of the WORKING-STORAGE SECTION quite different due to the lack of the 77 level. No restriction was placed on the size of the level numbers other than they must be less than 255. This allows for the standard practice of level skipping. In addition, it would not be dificult to make the 77 level perform in a normal manner. There is no difference in the way that the language considers an 01 level and a 77 level item, but the compatability with common usage would be very helpful to a COBOL programmer.

It is hoped that the results of this project are in a form that will allow others to use the compiler as a working system. It is recognized that many undiscovered problems will plague the initial users, but every effort has been made to describe what the system should do and to isolate the functions within the interpreter to facilitate changes.

APPENDIX A - MICRO-COBOL USERS MANUAL

This manual is written to explain the implimentation of HYPO-COBOL done at the Naval Postgraduate School for the Intel 8080 microcomputer running with CP/M (Control Program / Microcomputer). It is not intended that this manual take the place of the HYPO-COBOL specification but that it supply information on the manner in which this implimentation was done. There is no attempt to teach COBOL; however, someone who has a working knowledge of the language should be able to produce programs from the information contained in this manual.

This manual contains a brief overview of the justification for HYPO-COBOL and the organization of this implimentation. It contains a brief explanation of each of the constructs available in the language and shows samples of their use. It explains the interactions between the various parts of the compiler and interpreter and how they interface with the operating system. It also includes a list of references that might be useful to someone who wished to modify the compiler.

One of the major goals of this document is to explain how the operating system used effects the operation of the compiler. It is recognized that if the implimentation is to be useful it will need to be modified to run on other confi-

gurations of hardware and on other operating systems. Where it was possible, the interaction with the operating environment was insulated from the other parts of the program, but in the case of the file structure certain assumptions had to be made that could require modification.

ACKNOWLEDGEMENT

Any organization interested in reproducing the COBOL report and specifications in whole or in part, using ideas from this report as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce the following acknowledgment paragraphs in their entirety as part of the preface to any such publication. Any organization using a short passage from this document, such as in a book review, is requested to mention "COBOL" in acknowledgement of the source, but need not quote the acknowledgement.

> COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations. No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein

> FLOW-MATIC (trademark of Sperry Rand Cororation), programming for the Univac (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM comercial Translator Form No. F 28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell.

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

CONTENTS

I. HYPO-COBOL OVERVIEW

In order to provide a standard COBOL subset that could be implimented on a small computer system, the Department of the Navy has defined HYPO-COBOL. This definition is intended to give the minimum subset of the COBOL language that would be useable as a working product. This subset does not agree with the lowest level of COBOL as defined by the CODASYL group and in some cases includes only a portion of one of the COBOL levels as defined in the current standards. It is defined to include a portion of the NUCLEUS and both SEQUENTIAL I-O and RELATIVE I-O. A small portion of the DEBUG module was included along with some INTERPROGRAM COMMUNICATION instructions.

Where possible, short forms were included rather than long forms, and if two forms existed for the same instruction, only one was included. For example, the shortened PIC is used rather than the full word PICTURE. Also GO is not followed by the optional word TO. This does allow the definition to be a proper subset of the standard COBOL, but, at the same time, reduces the impact of the wordiness of COBOL on a small system.

As an exception to the general rule, PERFORM UNTIL was included from level 2 of the NUCLEUS in order to provide an additional control structure to support structured program-

ming techniques. Further information on HYPO-COBOL can be found in reference 6.

II. ORGANIZATION OF THE IMPLIMENTATION

The compiler is designed to run on an 8080 system in an interactive mode through the use of a teletype or console. It requires at least 12k of RAM memory and a mass storage device for reading and writing. The compiler is composed of two parts or passes, each of which reads a portion of the input file. Pass one reads the input program and builds the symbol table. At the end of the DATA DIVISION, pass one is overlayed by pass two which uses the symbol table to produce the code. The output code is written as it is produced to minimize the use of internal storage.

The first program of the interpreter builds the core image of the code and performs such functions as back-stuffing addresses. This first program loads the second program in and relenquishes control to the run time environment. The interpreter is controlled by a large case statement that decodes the instructions and performs the required actions.

As a tool for debugging the compiler a seperate program was created that will read the output code and translate the operations back into the mnemonics that are used in the second pass of the compiler. This "decode" program has been included with the other programs in order that anyone wishing to make changes to the output code or to the actions of

the interpreter can use this tool.

III. MICRO-COBOL ELEMENTS

This section contains a description of each element in the language and shows simple examples of its use. The following conventions are used in explaining the formats: Elements inclosed in broken braces < > are themselves complete entities and are described elsewhere in the manual. Elements inclosed in stacks of braces { } are choices, one of the elements which is be used. Elements inclosed in brackets [ ] are optional. All elements in capital letters are reserved words and must be spelled exactly.

User names are indicated as lower case. These names have been restricted to 12 characters in length. There are no restrictions in the compiler on what characters may be in a user name. Some restrictions do need to be made to assure that they are not taken as literal numbers when used in the DATA DIVISION. For example a record could be defined in the DATA DIVISION with the name 1234, but the command MOVE 1234 TO RECORD1 would result in the movement of the literal number not the data stored. The HYPO-COBOL description requires that each name start with a letter. This restriction was not implemented because it violates common programming practices.

The input to the compiler does not need to conform to standard COBOL format. Freeform input will be accepted as

the default condition. If desired, sequence numbers can be entered in the first six positions of each line. However, a toggle needs to be set to cause the compiler to ignore those lines.

ELEMENT:

IDENTIFICATION DIVISION Format

FORMAT:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. <comment>.
[AUTHOR. <comment>.]
[DATE-WRITTEN. <comment>.]
[SECURITY. <comment>.]
```

DESCRIPTION:

This division provides information for program identification for the reader. The order of the lines is fixed.

EXAMPLES:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.
AUTHOR. A S CRAIG.
```

ELEMENT:

ENVIRONMENT DIVISION Format

FORMAT:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. <comment> [DEBUGGING MODE].
OBJECT-COMPUTER. <comment>.
[INPUT-OUTPUT SECTION.
FILE-CONTROL.
   <file-control-entry> . . .
[I-O-CONTROL.
   SAME file-name-1 file-name-2 [file-name-3]
              [file-name-4] [file-name-5].  ]   ]
```

DESCRIPTION:

This division determines the external nature of a file. In the case of CP/M all of the files used can be accessed either sequentially or randomly except for variable length files which are sequential only. The debugging mode is also set by this section.

ELEMENT:

<file-control-entry>

FORMAT:

1.

```
SELECT file-name
    ASSIGN implementor-name
    [ORGANIZATION SEQUENTIAL]
    [ACCESS SEQUENTIAL].
```

2.

```
SELECT file-name
    ASSIGN implementor-name
    ORGANIZATION RELATIVE
    [ACCESS {SEQUENTIAL [RELATIVE data-name]}].
            {RANDOM RELATIVE data-name       }
```

DESCRIPTION:

The file-control-entry defines the type of file that the program expects to see. There is no difference on the diskette, but the type of reads and writes that are performed will differ. For CP/M the implementor name needs to conform to the normal specifications.

EXAMPLES:

```
SELECT CARDS
    ASSIGN CARD.FIL.


SELECT RANDOM-FILE
    ASSIGN A.RAN
    ORGANIZATION RELATIVE
    ACCESS RANDOM RELATIVE RAND-FLAG.
```

ELEMENT:

DATA DIVISION Format

FORMAT:

```
DATA DIVISION.
[FILE SECTION.
[FD file-name
    [BLOCK integer-1 RECORDS]
    [RECORD [integer-2 TO] integer-3]
    [LABEL RECORD {STANDARD}]
                  {OMITTED }
    [VALUE OF implementor-name-1 literal-1
         [implementor-name-2 literal-2] ... ].
[<record-description-entry>] ...] ...

[WORKING-STORAGE SECTION.
[<record-description-entry>] ... ]

[LINKAGE SECTION.
[<record-description-entry>] ... ]
```

DESCRIPTION:

This is the section that describes how the data is structured. There are no major differences from standard COBOL except for the following: 1. Label records make no sense on the diskette so no entry is

required. 2. The VALUE OF clause likewise has no meaning for CP/M. 3. The linkage section has not been implimented.

If a record is given two lengths as in RECORD 12 TO 128, the file is taken to be variable length and can only be accessed in the sequential mode. See the section on files for more information.

ELEMENT:

<comment>

FORMAT:

any string of characters

DESCRIPTION:

A comment is a string of characters. It may include anything other than a period followed by a blank or a reserved word, either of which terminate the string. Comments may be empty if desired, but the terminator is still required by the program.

EXAMPLES:

this is a comment

anotheroneallruntogether

8080b 16K

ELEMENT:

<data-description-entry> Format

FORMAT:

```
level-number {data-name}
             {FILLER   }

[REDEFINES data-name]

[PIC character-string]

[USAGE {COMP    }]
       {DISPLAY}

[SIGN {LEADING} [SEPARATE]]
      {TRAILING}

[OCCURS integer]

[SYNC [LEFT ]]
      [RIGHT]

[VALUE literal].
```

DESCRIPTION:

This statement describes the specific attributes of the data. Since the 8080 is a byte machine, there was no meaning to the SYNC clause, and thus it has not been implimented.

EXAMPLES:

```
01 CARD-RECORD.
   02 PART PIC X(5).
   02 NEXT-PART PIC 99V99 USAGE COMP.
   02 FILLER.
      03 NUMB PIC S9(3)V9 SIGN LEADING SEPARATE.
      03 LONG-NUMB 9(15).
      03 STRING REDEFINES LONG-NUMB PIC X(15).
   02 ARRAY PIC 99 OCCURS 100.
```

ELEMENT:

PROCEDURE DIVISION Format

FORMAT:

1.

```
PROCEDURE DIVISION [USING name1 [name2] ... [name5]].
section-name SECTION.
[paragraph-name. <sentence> [<sentence> ... ] ... ] ...
```

2.

```
PROCEDURE DIVISION [USING name1 [name2] ... [name5]].
paragraph-name. <sentence> [<sentence> ...] ...
```

DESCRIPTION:

As is indicated, if the program is to contain sections, then the first paragraph must be in a section. The USING option is part of the interprogram communication module and has not been implimented.

ELEMENT:

<sentence>

FORMAT:

<imperative-statement>

<conditional-statement>

ENTER verb

DESCRIPTION:

All sentences other than ENTER fall in one of the two main catigories. ENTER is part of the interprogram communication module.

ELEMENT:

<imperative-statement>

FORMAT:

The following verbs are always imperatives:

ACCEPT

CALL

CLOSE

DISPLAY

EXIT

GO

MOVE

OPEN

PERFORM

STOP

The following may be imperatives:

arithmetic verbs without the SIZE ERROR statement

and DELETE, WRITE, and REWRITE without the INVALID option.

ELEMENT:

<conditional-statements>

FORMAT:

IF

READ

arithmetic verbs with the SIZE ERROR statement

and DELETE, WRITE, and REWRITE with the INVALID option.

ELEMENT:

ACCEPT

FORMAT:

ACCEPT <identifier>

DESCRIPTION:

This statement reads up to 72 characters from the console. The usage of the item must be DISPLAY.

EXAMPLES:

```
ACCEPT IMMAGE
ACCEPT NUM(9)
```

ELEMENT:

ADD

FORMAT:

```
ADD {identifier} [{identifier-1}] TO identifier-2
    {literal    }  {literal     }

   [ROUNDED] [SIZE ERROR <imperative-statement>].
```

DESCRIPTION:

This instruction adds either one or two numbers to a third with the result being placed in the last location.

EXAMPLES:

```
ADD 10 TO NUMB1

ADD X Y TO Z ROUNDED.

ADD 100 TO NUMBER SIZE ERROR GO ERROR-LOC
```

ELEMENT:

CALL

FORMAT:

CALL literal [USING name1 [name2] ... [name5]]

DESCRIPTION:

CALL is not implimented.

ELEMENT:

CLOSE

FORMAT:

CLOSE file-name

DESCRIPTION:

Files must be closed if they have been written. However, the normal requirement to close an input file prior to the end of processing does not exist.

EXAMPLES:

```
CLOSE FILE1
CLOSE RANDFILE
```

ELEMENT:

DELETE

FORMAT:

DELETE record-name [INVALID <imperative-statement>]

DESCRIPTION:

This statement requires the record name, not the file name as in the standard form of the statement. Since there is no deletion mark in CP/M, this would normally result in the record still being readable. It is, therefore, filled with zeroes to indicate that it has been removed.

EXAMPLES:

DELETE RECORD1

ELEMENT:

DISPLAY

FORMAT:

```
DISPLAY {identifier} [{identifier-1}]
        {literal   }  {literal     }
```

DESCRIPTION:

This displays the contents of an identifier or displays a literal on the console. Usage must be DISPLAY. The maximum length of the display is 72 positions.

EXAMPLES:

```
DISPLAY MESSAGE-1
DISPLAY MESSAGE-3 10
DISPLAY 'THIS MUST BE THE END'
```

ELEMENT:

DIVIDE

FORMAT:

```
DIVIDE {identifier} into identifier-1 [ROUNDED]
       {literal    }
    [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

The result of the division is stored in identifier-1; any remainder is lost.

EXAMPLES:

```
DIVIDE NUMB INTO STORE
DIVIDE 25 INTO RESULT
```

ELEMENT:

ENTER

FORMAT:

ENTER language-name [routine-name]

DESCRIPTION:

This construct is not implimented.

ELEMENT:

EXIT

FORMAT:

EXIT [PROGRAM]

DESCRIPTION:

The EXIT command causes no action by the interpreter but allows for an empty paragraph for the construction of a common return point. The optional PROGRAM statement is not implimented as it is part of the interprogram communication module.

EXAMPLES:

```
RETURN.
    EXIT.
```

ELEMENT:

GO

FORMAT:

1.

GO procedure-name

2.

GO procedure-1 [procedure-2] ... procedure-20
DEPENDING identifier

DESCRIPTION:

The go command causes an unconditional branch to the routine specified. The second form causes a forward branch depending on the value of the contents of the identifier. The identifier must be a numeric integer value. There can be no more than 20 procedure names.

EXAMPLES:

GO READ-CARD.

GO READ1 READ2 READ3 DEPENDING READ-INDEX.

ELEMENT:

IF

FORMAT:

```
IF <condition> {imperative    } ELSE imperative-2
               {NEXT SENTENCE}
```

DESCRIPTION:

This is the standard COBOL IF statement. Note that there is no nesting of IF statements allowed since the IF statement is a conditional.

EXAMPLES:

```
IF A GREATER B ADD A TO C ELSE GO ERROR-ONE.

IF A NOT NUMERIC NEXT SENTENCE ELSE MOVE ZERO TO A.
```

ELEMENT:

MOVE

FORMAT:

```
MOVE {identifier-1} TO identifier-2
     {literal      }
```

DESCRIPTION:

The standard list of allowable moves applies to this action. As a space saving feature of this implimentation, all numeric moves go through the accumulators. This makes numeric moves slower than alpha-numeric moves, and where possible they should be avoided. Any move that involves picture clauses that are exactly the same can be accomplished as an alpha-numeric move if the elements are redefined as alpha-numeric; also all group moves are alpha-numeric.

EXAMPLES:

```
MOVE SPACE TO PRINT-LINE.
MOVE A(10) TO B(PTR).
```

ELEMENT:

MULTIPLY

FORMAT:

```
MULTIPLY {identifier} BY identifier-2 [ROUNDED]
         {literal    }

   [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

The multiply routine requires enough space to calculate the result with the full number of decimal digits prior to moving the result into identifier-2. This means that a number with 5 places after the decimal multiplied by a number with 6 places after the decimal will generate a number with 11 decimal places which would overflow if there were more than 7 digits before the decimal place.

EXAMPLES:

```
MULTIPLY X BY Y.

MULTIPLY A BY B(7) SIZE ERROR GO OVERFLOW.
```

ELEMENT:

OPEN

FORMAT:

```
OPEN {INPUT file-name }
     {OUTPUT file-name}
     {I-O file-name    }
```

DESCRIPTION:

These three types of opens have the exact same effect on the diskette. However, they do allow for internal checking of the other file actions. For example, a write to a file set open as input will cause a fatal error.

EXAMPLES:

```
OPEN INPUT CARDS.

OPEN OUTPUT REPORT-FILE.
```

ELEMENT:

PERFORM

FORMAT:

1.

PERFORM procedure-name [THRU procedure-name-2]

2.

PERFORM procedure-name [THRU procedure-name-2]

{identifier} TIMES
{integer   }

3.

PERFORM procedure-name [THRU procedure-name-2]

UNTIL <condition>

DESCRIPTION:

All three options are supported. Branching may be either forward or backward, and the procedures called may have perform statements in them as long as the end points do not coincide or overlap.

EXAMPLES:

PERFORM OPEN-ROUTINE.

PERFORM TOTALS THRU END-REPORT.

PERFORM SUM 10 TIMES.

PERFORM SKIP-LINE UNTIL PG-CNT GREATER 60.

ELEMENT:

READ

FORMAT:

1.

READ file-name INVALID <imperative-statement>

2.

READ file-name END <imperative-statement>

DESCRIPTION:

The invalid condition is only applicable to files in a random mode. All sequential files must have an END statement.

EXAMPLES:

READ CARDS END GO END-OF-FILE.

READ RANDOM-FILE INVALID MOVE SPACES TO REC-1.

ELEMENT:

REWRITE

FORMAT:

REWRITE file-name [INVALID <imperative>]

DESCRIPTION:

REWRITE is only valid for files that are open in the I-O mode. The INVALID clause is only valid for random files. This statement results in the current record being written back into the place that it was just read from. Note that this requires a file name not a record name.

EXAMPLES:

REWRITE CARDS.

REWRITE RAND-1 INVAID PERFORM ERROR-CHECK.

ELEMENT:

STOP

FORMAT:

```
STOP {RUN    }
     {literal}
```

DESCRIPTION:

This statement ends the running of the interpreter. If a literal is specified, then the literal is displayed on the console prior to termination of the program.

EXAMPLES:

```
STOP RUN.
STOP 1.
STOP "INVALID FINISH".
```

ELEMENT:

SUBTRACT

FORMAT:

```
SUBTRACT {identifier-1} [identifier-2] FROM identifier-3
         {literal-1   } [literal-2   ]

   [ROUNDED] [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

Identifier-3 is decremented by the value of identifier/literal one, and, if specified, identifier/literal two. The results are stored back in identifier-3. Rounding and size error options are available if desired.

EXAMPLES:

```
SUBTRACT 10 FROM SUB(12).
SUBTRACT A B FROM C ROUNDED.
```

ELEMENT:

WRITE

FORMAT:

1.

```
WRITE file-name [{BEFORE} ADVANCING {INTEGER}]
                 {AFTER }           {PAGE    }
```

2.

```
WRITE file-name INVALID <imperative-statement>
```

DESCRIPTION:

There is no printer on the 8080 system here, so the ADVANCING option is not implimented. The INVALID option only applies to random files.

EXAMPLES:

```
WRITE OUT-FILE.
WRITE RAND-FILE INVALID PERFORM ERROR-RECOV.
```

ELEMENT:

<condition>

FORMAT:

RELATIONAL CONDITION:

```
{identifier-1} [NOT] {GREATER} {identifier-2}
{literal-1}          {LESS   } {literal-2   }
                     {EQUAL  }
```

CLASS CONDITION:

```
identifier [NOT] {NUMERIC   }
                 {ALPHABETIC}
```

DESCRIPTION:

It is not valid to compare two literals. The class condition NUMERIC will allow for a sign if the identifier is signed numeric.

EXAMPLES:

```
A NOT LESS 10.

LINE GREATER "C".

NUMB1 NOT NUMERIC
```

ELEMENT:

Subscripting

FORMAT:

data-name (subscript)

DESCRIPTION:

Any item defined with an OCCURS many be referenced by a subscript. The subscript may be a literal integer, or it may be a data item that has been specified as an integer. If the subscript is signed, the sign must be positive at the time of its use.

EXAMPLES:

A(10)

ITEM(SUB)

IV. COMPILER TOGGLES

There are four toggles in the compiler. They are entered on the first line of the program as a dollar sign followed by the given letter. In each case the toggle reverses the default value.

$L -- list the input code on the screen as the program is compiled. Default is on. Error messages will be difficult to understand if this toggle is turned off, but if the interface device is a teletype, it may be desired in certain situations.

$S -- sequence numbers are in the first six positions of each record. Default is off.

$P -- list productions as they occur. Default is off.

$T -- list tokens from the scanner. Default is off.

V. RUN TIME CONVENTIONS

This section explains how to run the compiler on the current system. The compiler expects to see a file with a type of CBL as the input file. In general, the input is free form. If the input includes line numbers then the compiler must be notified by setting the appropriate toggle. The compiler is started by typing COBOL <file-name>. Where the file name is the system name of the input file. There is no interaction required to start the second part of the compiler. The output file will have the same file name as the input file, and will be given a file type of CIN. Any previous copies of the file will be erased.

The interpreter is started by typing CBLINT <file-name>. The first program is a loader, and it will display "LOAD FINISHED" to indicate successful completion. The run-time package will be brought in by the build program, and execution should continue without interuption.

VI. FILE INTERACTIONS WITH CP/M

The file structure that is expected by the program imposes some restrictions on the system. References 2 and 3 contain detailed information on the facilities of CP/M, and should be consulted for details. The information that has been included in this section is intended to explain where limitations exist and how the program interacts with the system.

All files in CP/M are on a random access device, and there is no way for the system to distinguish sequential files from files created in a random mode. This means that the various types of reads and writes are all valid to any file that has fixed length records. The restrictions of the ASSIGN statement do prevent a file from being open for both random and sequential actions during one program.

Each logical record is terminated by a carriage return and a line feed. In the case of variable length records, this is the only end mark that exists. This convention was addopted to allow the various programs which are used in CP/M to work with the files. Files created by the editor, for example, will generally be variable length files. This convention does remove the capability of reading variable length files in a random mode.

All of the physical records are assumed to be 128 bytes in length, and the program supplies buffer space for these records in addition to the logical records. Logical records may be of any desired length.

ERROR MESSAGES

COMPILER FATAL MESSAGES

BR Bad read -- disk error, no corrective action can be taken in the program.

CL Close error -- unable to close the output file.

MA Make error -- could not create the output file.

MO Memory overflow -- the code and constants generated will not fit in the alloted memory space.

OP Open error -- can not open the input file, or no such file present.

ST Symbol table overflow -- symbol table is too large for the allocated space.

WR Write error -- disk error, could not write a code record to the disk.

COMPILER WARNINGS

EL Extra levels -- only 10 levels are allowed.

FT File type -- the data element used in a read or write statement is not a file name.

IA Invalid access -- the specified options are not an allowable combination.

ID Identifier stack overflow -- more than 20 items in a GO TO -- DEPENDING statement.

IS Invalid subscript -- an item was subscripted but it was not defined by an OCCURS.

IT Invalid type -- the field types do not match for this statement.

LE Literal error -- a literal value was assigned to an item that is part of a group item previously assigned a value.

NF No file assigned -- there was no SELECT clause for this file.

NI Not implimented -- a production was used that is not implimented.

NN Non-numeric -- an invalid character was found in a numeric string.

NP No production -- no production exists for the cuurrent parser configuration; error recovery will automatically occur.

NV Numeric value -- a numeric value was assigned to a non-numeric item.

PC Picture clause -- an invalid character or set of characters exists in the picture clause.

PF Paragraph first -- a section header was produced after a paragraph header, which is not in a section.

R1 Redefine nesting -- a redefinition was made for an item which is part of a redefined item.

R2 Redefine length -- the length of the redefinition item was greater than the item that it redefined.

SE Scanner error -- the scanner was unable to read an identifier due to an invalid character.

SG Sign error -- either a sign was expected and not found, or a sign was present when not valid.

SL Significance loss -- the number assigned as a value is larger than the field defined.

TE Type error -- the type of a subscript index is not integer numeric.

VE Value error -- a value statement was assigned to an item in the file section.

INTERPRETER FATAL ERRORS

CL Close error -- the system was unable to close an output file.

ME Make error -- the system was unable to make an input file on the disk.

NF No file -- an input file could not be opened.

WI Write to input -- a write was attempted to an input file.

INTERPRETER WARNING MESSAGES

EM End mark -- a record that was read did not have a carriage return or a line feed in the expected location.

GD Go to depending -- the value of the depending indicator was greater than the number of available branch

addresses.

IC Invalid character -- an invalid character was loaded into an output field during an edited move. For example, a numeric character into an alphabetic-only field.

SI Sign Invalid -- the sign is not a "+" or a "-".

LIST OF REFERENCES

1. Craig, A. S. MICRO-COBOL an implementation of Navy Standard HYPO-COBOL for a microprocessor-based computer system, Masters Thesis, Naval Postgraduate School, March 1977.

2. Digital Research, An Introduction to CP/M Features and Facilities, 1976

3. Digital Research, CP/M Interface Guide, 1976.

4. Intel Corporation, 8008 and 8080 PL/M Programming Manual, 1975.

5. Intel Corperation, 8080 Simulator Software Package, 1974.

6. Software Development Division, ADPE Selection Office, Department of the Navy, HYPO-COBOL, April 1975.

7. Strutynski, Kathryn B. Information on the CP/M Interface Simulator, internally distributed technical note.

```
        /*       COBOL COMPILER - PART 1                   */


100H:  /*   LOAD POINT */

        /*    GLOBAL DECLARATIONS AND LITERALS    */

DECLARE LIT LITERALLY 'LITERALLY';
DECLARE
     BDOS             LIT         '5H', /* ENTRY TO OPERATING SYSTEM */
     MAX$MEMORY       LIT         '3100H',  /* TOP OF USEABLE MEMORY */
     INITIAL$POS      LIT         '2C00H',
     RDR$LENGTH       LIT         '255',
     PASS1$LEN        LIT         '46',
     BOOT             LIT         '0',
     CR               LIT         '13',
     LF               LIT         '10',
     QUOTE            LIT         '22H',
     POUND            LIT         '23H',
     TRUE             LIT         '1',
     FALSE            LIT         '0',
     FOREVER          LIT         'WHILE TRUE';




DECLARE MAXRNO LITERALLY '104',/* MAX READ COUNT */
        MAXLNO LITERALLY '129',/* MAX LOOK COUNT */
        MAXPNO LITERALLY '145',/* MAX PUSH COUNT */
        MAXSNO LITERALLY '234',/* MAX STATE COUNT */
        STARTS LITERALLY '1';/* START STATE */

DECLARE READ1 DATA(0,57,48,56,32,8,25,59,2,16,17,22,29,53,58,11,32,32,39
     ,38,34,44,9,19,32,37,6,33,3,14,15,18,20,32,28,49,32,1,42,38,36,43,1
     ,1,1,1,1,1,1,1,1,10,1,39,1,1,1,38,40,49,38,39,1,1,38,23,24,55,52,41
     ,35,46,1,7,50,1,32,1,32,32,45,1,32,1,32,1,32,47,37,4,26,32,54,40,1,1
     ,32,5,12,13,21,22,27,1,60,1,23,24,55,30,51);
DECLARE LOOK1 DATA(0,8,0,25,0,9,19,0,42,0,42,0,1,0,52,0,41,0,35,0,1,0,47
     ,0,4,0,54,0,40,0,35,46,60,0,1,0,32,0,1,0,1,0,11,0,60,0,7,0,32,0,32,0
     ,32,0);
DECLARE APPLY1 DATA(0,0,0,0,0,0,9,10,12,14,19,0,0,0,0,0,0,101,0,0,100,0
     ,0,0,0,0,0,97,0,27,0,0,0,69,0,91,92,0,0,91,92,0,0,0,0,13,17,0,102
     ,103,104,0,0,0,0,0,95,0,0,34,0,0,23,30,38,39,0,21,40,52,56,87,93,94
     ,0);
DECLARE READ2 DATA(0,65,57,64,154,26,37,67,21,30,31,33,39,61,66,27,234
     ,215,51,45,108,109,223,224,233,43,216,217,22,230,229,232,231,228,173
     ,172,169,9,226,47,196,195,7,8,11,13,15,2,3,105,14,158,4,50,20,12,18
     ,48,171,170,44,49,19,10,46,35,36,63,60,53,42,146,16,25,58,106,155
     ,148,155,155,55,150,155,152,155,157,155,56,193,23,208,234,62,52,206
     ,180,234,24,28,107,32,34,38,17,68,164,35,36,63,40,59);
DECLARE LOOK2 DATA(0,5,130,6,131,29,29,132,41,133,54,134,135,69,71,136
     ,72,137,73,138,139,80,84,140,86,198,88,141,89,142,184,184,184,91,189
     ,92,93,197,211,95,143,96,97,176,99,144,145,101,102,200,103,202,104
     ,188);
DECLARE APPLY2 DATA(0,0,77,111,112,147,79,114,81,82,83,78,76,117,75,156
     ,126,163,162,100,166,165,167,118,168,160,124,179,178,94,121,74,125
     ,120,119,187,187,186,98,192,192,191,194,113,183,128,129,127,205,205
     ,205,204,115,123,90,122,214,213,221,219,218,222,199,85,220,116,87
     ,110,70,174,209,207,182,182,181);
DECLARE INDEX1 DATA(0,1,2,3,4,5,6,7,8,4,4,24,4,24,4,13,14,24,109,4,15,16
     ,16,24,17,18,19,16,20,22,24,25,26,28,29,34,36,37,24,24,16,38,39,40
     ,42,43,44,45,46,47,48,49,16,50,38,51,16,52,53,54,55,56,57,58,60,61
     ,62,63,64,8,65,68,69,70,71,72,73,74,75,77,79,81,83,85,87,88,89,90,92
     ,93,94,8,8,16,95,97,97,15,103,104,105,109,24,24,24,1,3,5,8,10,12,14
     ,16,18,20,22,24,26,28,30,34,36,38,40,42,44,46,48,50,52,185,149,225
     ,227,227,190,151,153,203,159,210,161,175,212,201,177,1,2,3,3,4,4,5,5
     ,6,6,12,13,14,14,15,15,16,16,17,19,19,20,20,20,22,22,23,23,24,24,25
     ,25,26,26,27,29,29,31,32,32,33,33,35,38,38,33,33,39,39,39,39,39,42
     ,42,43,43,44,44,45,45,48,52,52,53,53,54,54,55,55,56,56,56,56,56,56
     ,56,56,58,58,58,59,59,61,61,61,61,61,62,67);
DECLARE INDEX2 DATA(0,1,1,1,1,1,1,1,5,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
     ,1,1,2,2,1,1,2,1,5,2,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
     ,1,1,1,2,1,1,1,1,1,5,3,1,1,1,1,1,1,1,2,2,2,2,2,2,1,1,1,2,1,1,1,5,5,1
     ,2,6,6,1,1,1,4,2,1,1,1,2,2,3,2,2,2,2,2,2,2,2,2,2,4,2,2,2,2,2,2,2,2
     ,2,2,5,6,29,41,54,69,71,72,73,80,84,88,89,96,99,101,3,9,3,0,3,0,3,0
     ,0,1,7,8,1,0,6,0,0,1,3,0,1,1,2,1,0,0,0,0,1,0,2,0,0,1,2,0,1,5,3,0,0,1
     ,4,0,0,0,1,2,1,2,2,2,0,2,3,0,3,0,0,1,4,0,0,1,0,0,0,0,1,1,1,1,2,2,1,1
     ,1,0,0,0,0,0,0,0,0,0,0,0,0);

     /* END OF TABLES */
DECLARE
     /* JOINT DECLARATIONS
     THESE ITEMS ARE DECLARED TOGETHER IN THIS SECTION
     IN ORDER TO FACILITATE THEIR BEING SAVED FOR
     THE SECOND PART OF THE COMPILER.
     */

     OUTPUT$FCB      (33) BYTE INITIAL(0,'          ','CIN',0,0,0,0),
     DEBUGGING       BYTE        INITIAL (FALSE),
     PRINT$PROD      BYTE        INITIAL(FALSE),
     PRINT$TOKEN     BYTE        INITIAL(FALSE),
     LIST$INPUT      BYTE        INITIAL (FALSE),
     SEQ$NUM         BYTE        INITIAL (FALSE),
     NEXT$SYM        ADDRESS,
     POINTER         ADDRESS     INITIAL (100H),
     NEXT$AVAILABLE  ADDRESS     INITIAL (2002H),
     MAX$INT$MEM     ADDRESS     INITIAL (3200H),
     FILES$SEC$END   BYTE        INITIAL (FALSE),
     FREE$STORAGE    ADDRESS     INITIAL (2500H),

     /* I O BUFFERS AND GLOBALS */
     IN$ADDR ADDRESS INITIAL (5CH),
     INPUT$FCB BASED IN$ADDR (33) BYTE,
     OUTPUT$PTR  ADDRESS,
     OUTPUT$BUFF (128) BYTE,
     OUTPUT$END ADDRESS,
     OUTPUT$CHAR BASED OUTPUT$PTR BYTE;
```

```
MON1: PROCEDURE (F,A);
    DECLARE F BYTE, A ADDRESS;
    GO TO BDOS;
END MON1;

MON2: PROCEDURE (F,A) BYTE;
    DECLARE F BYTE, A ADDRESS;
    GO TO BDOS;
END MON2;

PRINTCHAR: PROCEDURE (CHAR);
    DECLARE CHAR BYTE;
    CALL MON1 (2,CHAR);
END PRINTCHAR;

CRLF: PROCEDURE;
    CALL PRINTCHAR(CR);
    CALL PRINTCHAR(LF);
END CRLF;

PRINT: PROCEDURE (A);
    DECLARE A ADDRESS;
    CALL MON1 (9,A);
END PRINT;

PRINT$ERROR: PROCEDURE (CODE);
    DECLARE CODE ADDRESS;
    CALL CRLF;
    CALL PRINTCHAR(HIGH(CODE));
    CALL PRINTCHAR(LOW(CODE));
END PRINT$ERROR;

FATAL$ERROR: PROCEDURE(REASON);
    DECLARE REASON ADDRESS;
    CALL PRINT$ERROR(REASON);
    CALL TIME(10);
    GO TO BOOT;
END FATAL$ERROR;

OPEN: PROCEDURE;
    IF MON2 (15,IN$ADDR)=255 THEN CALL FATAL$ERROR('OP');
END OPEN;

MORE$INPUT: PROCEDURE BYTE;
    /* READS THE INPUT FILE AND RETURNS TRUE IF A RECORD
          WAS READ.  FALSE IMPLIES END OF FILE */
    DECLARE DCNT BYTE;
    IF (DCNT:=MON2(20,.INPUT$FCB))>1 THEN CALL FATAL$ERROR('BR');
    RETURN NOT(DCNT);
END MORE$INPUT;

MAKE: PROCEDURE;
    /* DELETES ANY EXISTING COPY OF THE OUTPUT FILE
          AND CREATES A NEW COPY*/
    CALL MON1(19,.OUTPUT$FCB);
    IF MON2(22,.OUTPUT$FCB)=255 THEN CALL FATAL$ERROR('MA');
END MAKE;

WRITE$OUTPUT: PROCEDURE;
    /* WRITES OUT A BUFFER */
    CALL MON1(26,.OUTPUT$BUFF);     /* SET DMA */
    IF MON2(21,.OUTPUT$FCB)<>0 THEN CALL FATAL$ERROR('WR');
    CALL MON1(26,80H);     /* RESET DMA */
END WRITE$OUTPUT;

MOVE: PROCEDURE(SOURCE, DESTINATION, COUNT);
    /* MOVES FOR THE NUMBER OF BYTES SPECIFIED BY COUNT */
    DECLARE (SOURCE,DESTINATION) ADDRESS,
    (S$BYTE BASED SOURCE, D$BYTE BASED DESTINATION, COUNT) BYTE;
    DO WHILE (COUNT:=COUNT - 1) <> 255;
          D$BYTE=S$BYTE;
          SOURCE=SOURCE +1;
          DESTINATION = DESTINATION + 1;
    END;
END MOVE;

FILL: PROCEDURE(ADDR,CHAR,COUNT);
    /* MOVES CHAR INTO ADDR FOR COUNT BYTES */
    DECLARE ADDR ADDRESS,
    (CHAR,COUNT,DEST BASED ADDR) BYTE;
    DO WHILE (COUNT:=COUNT -1)<>255;
          DEST=CHAR;
          ADDR=ADDR + 1;
    END;
END FILL;

    /*    *   *   *   *   *   SCANNER LITS *   *   *   *   */
DECLARE
    LITERAL          LIT        '15',
    INPUT$STR        LIT        '32',
    PERIOD           LIT        '1',
    INVALID          LIT        '0';


    /*   * * * * SCANNER TABLES * * * *    */
DECLARE TOKEN$TABLE DATA
    /* CONTAINS THE TOKEN NUMBER ONE LESS THAN THE FIRST RESERVED WORD
    FOR EACH LENGTH OF WORD   */
    (0,0,1,4,5,15,22,32,38,44,47,49,51,55,56,57),

TABLE DATA('FD','OF','TO','PIC','COMP','DATA','FILE'
    ,'LEFT','MODE','SAME','SIGN','SYNC','ZERO','BLOCK','LABEL'
    ,'QUOTE','RIGHT','SPACE','USAGE','VALUE','ACCESS','ASSIGN'
    ,'AUTHOR','FILLER','OCCURS','RANDOM','RECORD','SELECT'
    ,'DISPLAY','LEADING','LINKAGE','OMITTED','RECORDS'
    ,'SECTION','DIVISION','RELATIVE','SECURITY','SEPARATE','STANDARD'
    ,'TRAILING','DEBUGGING','PROCEDURE','REDEFINES'
    ,'PROGRAM-ID','SEQUENTIAL','ENVIRONMENT','I-O-CONTROL'
    ,'DATE-WRITTEN','FILE-CONTROL','INPUT-OUTPUT','ORGANIZATION'
```

```
1        ,'CONFIGURATION','IDENTIFICATION','OBJECT-COMPUTER'
1        ,'SOURCE-COMPUTER','WORKING-STORAGE'),
1
1   OFFSET  (16) ADDRESS
1        /*  NUMBER OF BYTES TO INDEX INTO THE TABLE FOR EACH LENGTH */
1        INITIAL (0,0,0,6,9,45,80,128,170,218,245,265,
1             287,335,348,362),
1
1   WORD$COUNT DATA
1        /* NUMBER OF WORDS OF EACH SIZE */
1        (0,0,3,1,9,7,8,6,6,3,2,2,4,1,1,3),
1
1
1        MAX$LEN            LIT          '16',
1        ADD$END            DATA         ('PROCEDURE '),
1        LOCKED             BYTE         INITIAL (0),
1        HOLD               BYTE,
1        BUFFER$END         ADDRESS      INITIAL   (100H),
1        NEXT               BASED         POINTER  BYTE,
1        INBUFF             LIT          '80H',
1        CHAR               BYTE,
1        ACCUM$LENG         LIT          '50',
1        ACCUM              BYTE,
1        P$ACCUM            (ACCUM$LENG)        BYTE,
1        DISPLAY            BYTE         INITIAL (0),
1        DISPLAY$REST       (73)         BYTE,
1        TOKEN              BYTE;            /*RETURNED FROM SCANNER */
1
1
1        /* * * * *    PROCEDURES USED BY THE SCANNER * * * */
1
1   NEXT$CHAR: PROCEDURE BYTE;
2        IF LOCKED THEN
2        DO;
2             LOCKED=FALSE;
3             RETURN (CHAR:=HOLD);
3        END;
2        IF (POINTER:=POINTER + 1) >= BUFFER$END THEN
2        DO;
2           IF NOT MORE$INPUT THEN
3             DO;
3                 BUFFER$END=.MEMORY;
4                 POINTER=.ADD$END;
4             END;
3             ELSE POINTER=INBUFF;
3        END;
2        RETURN (CHAR:=NEXT);
2   END NEXT$CHAR;
1
1   GET$CHAR: PROCEDURE;
2        /* THIS PROCEDURE IS CALLED WHEN A NEW CHAR IS NEEDED WITHOUT
2        THE DIRECT RETURN OF THE CHARACTER*/
2        CHAR=NEXT$CHAR;
2   END GET$CHAR;
1
1   DISPLAY$LINE: PROCEDURE;
2        IF NOT LIST$INPUT THEN RETURN;
2        DISPLAY(DISPLAY + 1) = '$';
2        CALL PRINT(.DISPLAY$REST);
2        DISPLAY=0;
2   END DISPLAY$LINE;
1
1
1   LOAD$DISPLAY: PROCEDURE;
2        IF DISPLAY < 72 THEN
2             DISPLAY(DISPLAY:=DISPLAY + 1) = CHAR;
2        CALL GET$CHAR;
2   END LOAD$DISPLAY;
1
1   PUT: PROCEDURE;
2        IF ACCUM < ACCUM$LENG THEN
2        ACCUM(ACCUM:=ACCUM+1)=CHAR;
2        CALL LOAD$DISPLAY;
2   END PUT;
1
1   EAT$LINE: PROCEDURE;
2        DO WHILE CHAR<>CR;
2             CALL LOAD$DISPLAY;
3        END;
2   END EAT$LINE;
1
1   GET$NO$BLANK: PROCEDURE;
2        DECLARE (N,I) BYTE;
2        DO FOREVER;
2             IF CHAR = ' ' THEN CALL LOAD$DISPLAY;
3             ELSE
3             IF CHAR=CR THEN
3             DO;
3                  CALL DISPLAY$LINE;
4                  IF SEQ$NUM THEN N=8; ELSE N=2;
4                  DO I = 1 TO N;
4                       CALL LOAD$DISPLAY;
5                  END;
4                  IF CHAR = '*' THEN CALL EAT$LINE;
4                  ELSE
4                  IF CHAR = ':' THEN
4                       DO;
5                            IF NOT DEBUGGING THEN CALL EAT$LINE;
5                            ELSE CALL LOAD$DISPLAY;
5                       END;
4             END;
3             ELSE
3             RETURN;
3        END;    /* END OF DO FOREVER */
2   END GET$NO$BLANK;
1
1   SPACE: PROCEDURE BYTE;
2        RETURN (CHAR=' ') OR (CHAR=CR);
2   END SPACE;
1
```

```
1    DELIMITER: PROCEDURE BYTE;
2         /* CHECKS FOR A PERIOD FOLLOWED BY A SPACE OR CR*/
2         IF CHAR <> '.' THEN RETURN FALSE;
2         HOLD=NEXT$CHAR;
2         LOOKED=TRUE;
2         IF SPACE THEN
2         DO;
3              CHAR = '.';
3              RETURN TRUE;
3         END;
2         CHAR='.';
2         RETURN FALSE;
2    END DELIMITER;
1
1    END$OF$TOKEN: PROCEDURE BYTE;
2         RETURN SPACE OR  DELIMITER;
2    END END$OF$TOKEN;
1
1    GET$LITERAL: PROCEDURE BYTE;
2         CALL LOAD$DISPLAY;
2         DO FOREVER;
2              IF CHAR= QUOTE THEN
3              DO;
3                   CALL LOAD$DISPLAY;
4                   RETURN LITERAL;
4              END;
3              CALL PUT;
3         END;
2    END GET$LITERAL;
1
1
1    LOOK$UP: PROCEDURE BYTE;
2         DECLARE POINT ADDRESS,
2         (HERE BASED POINT,I) BYTE;
2
2         MATCH: PROCEDURE BYTE;
3              DECLARE J BYTE;
3              DO J=1 TO ACCUM;
3                   IF HERE(J - 1) <> ACCUM(J) THEN RETURN FALSE;
4              END;
3              RETURN TRUE;
3         END MATCH;
2
2         POINT=OFFSET(ACCUM)+ .TABLE;
2         DO I=1 TO WORD$COUNT(ACCUM);
3              IF MATCH THEN RETURN I;
3              POINT = POINT + ACCUM;
3         END;
2         RETURN FALSE;
2    END LOOK$UP;
1
1    RESERVED$WORD: PROCEDURE BYTE;
2         /* RETURNS THE TOKEN NUMBER OF A RESERVED WORD IF THE CONTENTS OF
2         THE ACCUMULATOR IS A RESERVED WORD, OTHERWISE RETURNS ZERO */
2         DECLARE VALUE BYTE;
2         DECLARE NUMB BYTE;
2              IF ACCUM > MAX$LEN THEN RETURN 0;
2              IF (NUMB:=TOKEN$TABLE(ACCUM))=0 THEN RETURN 0;
2              IF (VALUE:=LOOK$UP)=0 THEN RETURN 0;
2              RETURN (NUMB + VALUE);
2    END RESERVED$WORD;
1
1    GET$TOKEN: PROCEDURE BYTE;
2         ACCUM=0;
2         CALL GET$NO$BLANK;
2         IF CHAR=QUOTE THEN RETURN GET$LITERAL;
2         IF DELIMITER THEN
2         DO;
3              CALL PUT;
3              RETURN PERIOD;
3         END;
2         DO FOREVER;
3              CALL PUT;
3              IF END$OF$TOKEN THEN RETURN INPUT$STR;
3         END; /* OF DO FOREVER */
2    END GET$TOKEN;
1
1
1
1    SCANNER: PROCEDURE;
2         DECLARE CHECK BYTE;
2         DO FOREVER;
3              IF(TOKEN:=GET$TOKEN) = INPUT$STR THEN
3                   IF (CHECK:=RESERVED$WORD) <> 0 THEN TOKEN=CHECK;
3              IF TOKEN <> 0 THEN RETURN;
3              CALL PRINT$ERROR ('SE');
3              DO WHILE NOT END$OF$TOKEN;
4                   CALL GET$CHAR;
4              END;
3         END;
2    END SCANNER;
1
1
1    PRINT$ACCUM: PROCEDURE;
2         ACCUM(ACCUM+1)='$';
2         CALL PRINT(.R$ACCUM);
2    END PRINT$ACCUM;
1
1    PRINT$NUMBER: PROCEDURE(NUMB);
2         DECLARE(NUMB,I,CNT,K) BYTE, J DATA(100,10);
2         DO I=0 TO 1;
3              CNT=0;
3              DO WHILE NUMB >= (K:=J(I));
4                   NUMB=NUMB - K;
4                   CNT=CNT + 1;
4              END;
3              CALL PRINTCHAR('0' + CNT);
3         END;
2         CALL PRINTCHAR('0' + NUMB);
2    END PRINT$NUMBER;
```

```
1
1
1
1   INIT$SCANNER: PROCEDURE;
2       /*     INITIALIZE FOR INPUT - OUTPUT OPERATIONS     */
2       CALL MOVE (.'CBL', IN$ADDR + 9, 3);
2       CALL FILL(IN$ADDR + 12,0,5);
2       CALL OPEN;
2       CALL MOVE(INADDR,.OUTPUT$FCB,9);
2       OUTPUT$END=(OUTPUT$PTR:=.OUTPUT$BUFF - 1) + 128;
2       CALL MAKE;
2       CALL GET$CHAR;      /* PRIME THE SCANNER */
2       DO WHILE CHAR = '$';
2           IF NEXTCHAR = 'L' THEN LIST$INPUT=NOT LIST$INPUT;
3           ELSE IF CHAR ='S' THEN SEQ$NUM= NOT SEQ$NUM;
3           ELSE IF CHAR = 'P' THEN PRINT$PROD = NOT PRINT$PROD;
3           ELSE IF CHAR = 'T' THEN PRINT$TOKEN = NOT PRINT$TOKEN;
3           CALL GET$CHAR;
3           CALL GET$NO$BLANK;
3       END;
2   END INIT$SCANNER;
1
1       /*  *  *  *  END OF SCANNER PROCEDURES  *  *  *  */
1
1
1       /*  *  *  *  *  SYMBOL TABLE DECLARATIONS * * *  */
1
1   DECLARE
1
1   CUR$SYM              ADDRESS,         /*SYMBOL BEING ACCESSED*/
1   SYMBOL               BASED CUR$SYM  BYTE,
1   SYMBOL$ADDR          BASED CUR$SYM  ADDRESS,
1   NEXT$SYM$ENTRY       BASED NEXT$SYM ADDRESS,
1   HASH$PTR             ADDRESS,
1   DISPLACEMENT         LIT              '12',
1   HASH$MASK            LIT              '3FH',
1   S$TYPE               LIT              '2',
1   OCCURS               LIT              '11',
1   ADDR2                LIT              '4',
1   P$LENGTH             LIT              '3',
1   S$LENGTH             LIT              '3',
1   LEVEL                LIT              '10',
1   LOCATION             LIT              '2',
1   REL$ID               LIT              '5',
1   START$NAME           LIT              '11',   /*1 LESS*/
1   MAX$ID$LEN           LIT              '12';
1
1       /*  *  *  *  *  TYPE LITERALS * * * * * * * */
1
1   DECLARE
1   SEQUENTIAL      LIT       '1',
1   RANDOM          LIT       '2',
1   SEQ$RELATIVE    LIT       '3',
1   VARIABLE$LENG   LIT       '4',
1   GROUP           LIT       '6',
1   COMP            LIT       '21';
1
1       /*  *  *  *  SYMBOL TABLE ROUTINES  *  *  *  */
1
1   INIT$SYMBOL: PROCEDURE;
2       CALL FILL (FREE$STORAGE,0,130);
2       /* INITIALIZE HASH TABLE AND FIRST COLLISION FIELD */
2       NEXT$SYM=FREE$STORAGE+128;
2       NEXT$SYM$ENTRY=0;
2   END INIT$SYMBOL;
1
1   GET$P$LENGTH: PROCEDURE BYTE;
2       RETURN SYMBOL(P$LENGTH);
2   END GET$P$LENGTH;
1
1   SET$ADDRESS: PROCEDURE(ADDR);
2   DECLARE ADDR ADDRESS;
2       SYMBOL$ADDR(LOCATION)=ADDR;
2   END SET$ADDRESS;
1
1   GET$ADDRESS: PROCEDURE ADDRESS;
2       RETURN SYMBOL$ADDR(LOCATION);
2   END GET$ADDRESS;
1
1   GET$TYPE: PROCEDURE BYTE;
2       RETURN SYMBOL(S$TYPE);
2   END GET$TYPE;
1
1   SET$TYPE: PROCEDURE(TYPE);
2       DECLARE TYPE BYTE;
2       SYMBOL(S$TYPE)=TYPE;
2   END SET$TYPE;
1
1   OR$TYPE: PROCEDURE(TYPE);
2       DECLARE TYPE BYTE;
2       SYMBOL(S$TYPE)=TYPE OR GET$TYPE;
2   END OR$TYPE;
1
1   GET$LEVEL: PROCEDURE BYTE;
2       RETURN SHR(SYMBOL(LEVEL),4);
2   END GET$LEVEL;
1
1   SET$LEVEL: PROCEDURE (LVL);
2       DECLARE LVL BYTE;
2       SYMBOL(LEVEL)=SHL(LVL,4) OR SYMBOL(LEVEL);
2   END SET$LEVEL;
1
1   GET$DECIMAL: PROCEDURE BYTE;
2       RETURN SYMBOL(LEVEL) AND 0FH;
2   END GET$DECIMAL;
1
1   SET$DECIMAL: PROCEDURE (DEC);
2       DECLARE DEC BYTE;
2       SYMBOL(LEVEL) = DEC OR SYMBOL(LEVEL);
2   END SET$DECIMAL;
```

```
SET$S$LENGTH: PROCEDURE(HOW$LONG);
     DECLARE HOW$LONG ADDRESS;
     SYMBOL$ADDR(S$LENGTH) = HOW$LONG;
END SET$S$LENGTH;

GET$S$LENGTH: PROCEDURE ADDRESS;
     RETURN SYMBOL$ADDR(S$LENGTH);
END GET$S$LENGTH;


SET$ADDR2: PROCEDURE (ADDR);
     DECLARE ADDR ADDRESS;
SYMBOL$ADDR(ADDR2)=ADDR;
END SET$ADDR2;

GET$ADDR2: PROCEDURE ADDRESS;
     RETURN SYMBOL$ADDR(ADDR2);
END GET$ADDR2;

SET$OCCURS: PROCEDURE(OCCUR);
     DECLARE OCCUR BYTE;
     SYMBOL(OCCURS)=OCCUR;
END SET$OCCURS;

GET$OCCURS: PROCEDURE BYTE;
     RETURN SYMBOL (OCCURS);
END GET$OCCURS;

     /*  *  *  *  PARSER DECLARATIONS  *  *  *  */
DECLARE
INT                    LIT          '63',      /* CODE FOR INITIALIZE */
SCD                    LIT          '66',      /* CODE FOR SET CODE START */
PSTACKSIZE              LIT          '30',      /* SIZE OF PARSE STACKS*/
STATESTACK             (PSTACKSIZE)    BYTE,  /* SAVED STATES */
VALUE                  (PSTACKSIZE)   ADDRESS,     /* TEMP VALUES */
VARC                   (51)           BYTE,     /*TEMP CHAR STORE*/
ID$STACK               (10)       ADDRESS    INITIAL (0),
ID$STACK$PTR           BYTE      INITIAL(0),
HOLD$LIT               BYTE,
REST$HOLD$LIT          (ACCUM$LENG)    BYTE,
HOLD$SYM               ADDRESS,
PENDING$LITERAL        BYTE INITIAL(FALSE),
PENDING$LIT$ID         ADDRESS,
REDEF                  BYTE        INITIAL (FALSE),
REDEF$ONE              ADDRESS,
REDEF$TWO              ADDRESS,
TEMP$HOLD              ADDRESS,
TEMP$TWO               ADDRESS,
COMPILING              BYTE    INITIAL(TRUE),
SP                     BYTE    INITIAL (255),
MP                     BYTE,
MPP1                   BYTE,
NOLOOK                 BYTE    INITIAL(TRUE),
(I,J,K)                BYTE,        /*INDICIES FOR THE PARSER*/
STATE                  BYTE    INITIAL(STARTS);

     /*  *  *  *  PARSER ROUTINES  *  *  *  *  */

BYTE$OUT: PROCEDURE(ONE$BYTE);
     /* THIS PROCEDURE WRITES ONE BYTE OF OUTPUT ONTO THE DISK
     IF REQUIRED THE OUTPUT BUFFER IS DUMPED TO THE DISK */
     DECLARE ONE$BYTE BYTE;
     IF (OUTPUT$PTR:=OUTPUT$PTR + 1)> OUTPUT$END THEN
     DO;
          CALL WRITE$OUTPUT;
          OUTPUT$PTR=.OUTPUT$BUFF;
     END;
     OUTPUT$CHAR=ONE$BYTE;
END BYTE$OUT;

STRING$OUT: PROCEDURE (ADDR,COUNT);
     DECLARE (ADDR,I,COUNT) ADDRESS, (CHAR BASED ADDR) BYTE;
     DO I=1 TO COUNT;
          CALL BYTE$OUT(CHAR);
          ADDR=ADDR+1;
     END;
END STRING$OUT;

ADDR$OUT: PROCEDURE(ADDR);
     DECLARE ADDR ADDRESS;
     CALL BYTE$OUT(LOW(ADDR));
     CALL BYTE$OUT(HIGH(ADDR));
END ADDR$OUT;

FILL$STRING: PROCEDURE(COUNT,CHAR);
     DECLARE (I,COUNT) ADDRESS, CHAR BYTE;
     DO I=1 TO COUNT;
          CALL BYTE$OUT(CHAR);
     END;
END FILL$STRING;

START$INITIALIZE: PROCEDURE(ADDR,CNT);
     DECLARE (ADDR,CNT) ADDRESS;
     CALL BYTE$OUT(INT);
     CALL ADDR$OUT(ADDR);
     CALL ADDR$OUT(CNT);
END START$INITIALIZE;

BUILD$SYMBOL: PROCEDURE(LEN);
     DECLARE LEN BYTE, TEMP ADDRESS;
     TEMP=NEXT$SYM;
     IF (NEXT$SYM:=.SYMBOL(LEN:=LEN+DISPLACEMENT))
          > MAX$MEMORY THEN CALL FATAL$ERROR('ST');
     CALL FILL (TEMP,0,LEN);
END BUILD$SYMBOL;
```

```
MATCH: PROCEDURE ADDRESS;
     /* CHECKS AN IDENTIFIER TO SEE IF IT IS IN THE SYMBOL
     TABLE.  IF IT IS PRESENT, CUR$SYM IS SET FOR ACCESS.
     OTHERWISE A NEW ENTRY IS MADE AND THE PRINT NAME
     IS ENTERED.  ALL NAMES ARE TRUNCATED TO MAX$ID$LEN*/
     DECLARE (POINT,COLLISION BASED POINT) ADDRESS,
     (HOLD,I) BYTE;
     IF VARC>MAX$ID$LEN
           THEN VARC = MAX$ID$LEN;
           /* TRUNCATE IF REQUIRED */
     HOLD = 0;
     DO I=1 TO VARC;     /* CALCULATE HASH CODE */
           HOLD=HOLD + VARC(I);
     END;
     POINT=FREE$STORAGE + SHL((HOLD AND HASH$MASK),1);
     DO FOREVER;
          IF COLLISION=0 THEN
           DO;
                 CUR$SYM,COLLISION=NEXT$SYM;
                 CALL BUILD$SYMBOL(VARC);
                 /* LOAD PRINT NAME */
                 SYMBOL(P$LENGTH)=VARC;
                 DO I = 1 TO VARC;
                       SYMBOL(START$NAME + I)=VARC(I);
                 END;
                 RETURN CUR$SYM;
           END;
           ELSE
           DO;
                 CUR$SYM=COLLISION;
                 IF (HOLD:=GET$P$LENGTH)=VARC THEN
                 DO;
                       I=1;
                       DO WHILE SYMBOL(START$NAME + I)= VARC(I);
                        IF (I:=I+1)>HOLD THEN RETURN (CUR$SYM:=COLLISION);
                       END;
                 END;
          END;
          POINT=COLLISION;
     END;
END MATCH;

ALLOCATE: PROCEDURE(BYTES$REQ) ADDRESS;
     /* THIS ROUTINE CONTROLS THE ALLOCATION OF SPACE
     IN THE MEMORY OF THE INTERPRETER.   */

     DECLARE (HOLD,BYTES$REQ) ADDRESS;
     HOLD=NEXT$AVAILABLE;
     IF (NEXT$AVAILABLE:=NEXT$AVAILABLE + BYTES$REQ)>MAX$INT$MEM
           THEN CALL FATAL$ERROR('MO');
     RETURN HOLD;
END ALLOCATE;

SET$REDEF: PROCEDURE(OLD,NEW);
     DECLARE (OLD,NEW) ADDRESS;
     IF (REDEF:=NOT REDEF) THEN
     DO;
           REDEF$ONE=OLD;
           REDEF$TWO=NEW;
     END;
     ELSE CALL PRINT$ERROR('R1');
END SET$REDEF;

SET$CUR$SYM: PROCEDURE;
     CUR$SYM=ID$STACK(ID$STACK$PTR);
END SET$CUR$SYM;

STACK$LEVEL: PROCEDURE BYTE;
     CALL SET$CUR$SYM;
     RETURN GET$LEVEL;
END STACK$LEVEL;

LOAD$LEVEL: PROCEDURE;
     DECLARE HOLD ADDRESS;

     LOAD$REDEF$ADDR: PROCEDURE;
           CUR$SYM=REDEF$ONE;
           HOLD=GET$ADDRESS;
     END LOAD$REDEF$ADDR;

     IF ID$STACK<>0 THEN
     DO;
           IF VALUE(SP-2)=0 THEN
           DO;
                 CALL SET$CUR$SYM;
                 HOLD=GET$S$LENGTH + GET$ADDRESS;
           END;
           ELSE CALL LOAD$REDEF$ADDR;
           IF (ID$STACK$PTR:=ID$STACK$PTR+1)>9 THEN
           DO;
                 CALL PRINT$ERROR('EL');
                 ID$STACK$PTR=9;
           END;
     END;
     ELSE HOLD=NEXT$AVAILABLE;
     ID$STACK(ID$STACK$PTR)=VALUE(MPP1);
     CALL SET$CUR$SYM;
     CALL SET$ADDRESS(HOLD);
END LOAD$LEVEL;
```

```
REDEF$OR$VALUE: PROCEDURE;
    DECLARE HOLD ADDRESS,
        (DEC,K,J,SIGN) BYTE;
    IF REDEF THEN
    DO;
        IF REDEF$TWO=CUR$SYM THEN
        DO;
            HOLD=GET$S$LENGTH;
            CUR$SYM=REDEF$ONE;
            IF HOLD>GET$S$LENGTH THEN
            DO;
                CALL PRINT$ERROR('R2');
                HOLD=GET$S$LENGTH;
                CUR$SYM=REDEF$ONE;
                CALL SET$S$LENGTH(HOLD);
            END;
            REDEF=FALSE;
        END;
    END;
    ELSE IF PENDING$LITERAL=0 THEN RETURN;
    IF PENDING$LIT$ID<>ID$STACK$PTR THEN RETURN;
    CALL START$INITIALIZE(GET$ADDRESS,HOLD:=GET$S$LENGTH);
    IF PENDING$LITERAL>2 THEN
    DO;
        IF PENDING$LITERAL=3 THEN CHAR='0';
        ELSE IF PENDING$LITERAL=4 THEN CHAR=' ';
        ELSE CHAR=QUOTE;
        CALL FILL$STRING(HOLD,CHAR);
    END;
    ELSE IF PENDING$LITERAL = 2 THEN
    DO;
        IF HOLD <= HOLD$LIT THEN
            CALL STRING$OUT(.REST$HOLD$LIT,HOLD);
        ELSE DO;
            CALL STRING$OUT(.REST$HOLD$LIT,HOLD$LIT);
            CALL FILL$STRING(HOLD - (HOLD$LIT + 1),' ');
        END;
    END;
    ELSE DO;
        /* THE NUMBER HANDELER */
        DECLARE (DEC,MINUS$SIGN,I,J,LIT$DEC,N$LENGTH,
            NUM$BEFORE,NUM$AFTER, TYPE) BYTE, ZONE LIT '10H';

        IF((TYPE:=GET$TYPE)<16) OR (TYPE>20) THEN
            CALL PRINT$ERROR('NV');
        N$LENGTH=GET$S$LENGTH;
        DEC=GET$DECIMAL;
        MINUS$SIGN=FALSE;
        IF REST$HOLD$LIT='-' THEN
        DO;
            MINUS$SIGN=TRUE;
            J=1;
        END;
        ELSE IF REST$HOLD$LIT='+' THEN J=1;
        ELSE J=0;
        LIT$DEC=0;
        DO I=1 TO HOLD$LIT;
            IF HOLD$LIT(I)='.' THEN LIT$DEC=I;
        END;
        IF LIT$DEC=0 THEN
        DO;
            NUM$BEFORE=REST$HOLD$LIT-J;
            NUM$AFTER=0;
        END;
        ELSE DO;
            NUM$BEFORE=LIT$DEC -J-1;
            NUM$AFTER=REST$HOLD$LIT - LIT$DEC;
        END;
        IF (I:=N$LENGTH - DEC)<NUM$BEFORE THEN
            CALL PRINT$ERROR('SL');
        IF I>NUM$BEFORE THEN
        DO;
            I=I-NUM$BEFORE;
            IF MINUS$SIGN THEN
            DO;
                I=I-1;
                CALL BYTE$OUT('0' + ZONE);
            END;
            CALL FILL$STRING(I,'0');
        END;
        ELSE IF MINUS$SIGN THEN REST$HOLD$LIT(J)=REST$HOLD$LIT(J)+ZONE;
        CALL STRING$OUT(.REST$HOLD$LIT + J, NUM$BEFORE);
        IF NUM$AFTER > DEC THEN NUM$AFTER = DEC;
        CALL STRING$OUT(.REST$HOLD$LIT + LIT$DEC, NUM$AFTER);
        IF (I:=DEC - NUM$AFTER)<>0 THEN
            CALL FILL$STRING(I,'0');
    END;
    PENDING$LITERAL=0;
END REDEF$OR$VALUE;

REDUCE$STACK: PROCEDURE;
    DECLARE HOLD$LENGTH ADDRESS;
    CALL SET$CUR$SYM;
    CALL REDEF$OR$VALUE;
    HOLD$LENGTH=GET$S$LENGTH;
    IF GET$TYPE > 128 THEN
    DO;
        HOLD$LENGTH=HOLD$LENGTH * GET$OCCURS;
    END;
    ID$STACK$PTR=ID$STACK$PTR - 1;
    CALL SET$CUR$SYM;
    CALL SET$S$LENGTH(GET$S$LENGTH + HOLD$LENGTH);
    CALL SET$TYPE(GROUP);
END REDUCE$STACK;
```

```
1    END$OF$RECORD: PROCEDURE;
2         DO WHILE ID$STACK$PTR<>0;
2                  CALL REDUCE$STACK;
3         END;
2         CALL SET$CLR$SYM;
2         CALL REDEF$OR$VALUE;
2         ID$STACK=0;
2         TEMP$HOLD=ALLOCATE(TEMP$TWO:=GET$S$LENGTH);
2    END  END$OF$RECORD;
1
1    CONVERT$INTEGER: PROCEDURE;
2         DECLARE INTEGER ADDRESS;
2         INTEGER=0;
2         DO I = 1 TO VARC;
2              INTEGER=SHL(INTEGER,3)+SHL(INTEGER,1)+(VARC(I)-'0');
3         END;
2         VALUE(SP)=INTEGER;
2    END CONVERT$INTEGER;
1
1    OR$VALUE: PROCEDURE(PTR,ATTRIB);
2         DECLARE PTR BYTE, ATTRIB ADDRESS;
2         VALUE(PTR)=VALUE(PTR) OR ATTRIB;
2    END OR$VALUE;
1
1    BUILD$FCB: PROCEDURE;
2         DECLARE TEMP ADDRESS;
2        .DECLARE BUFFER(11) BYTE, (CHAR, I, J) BYTE;
2         CALL FILL(.BUFFER,' ',11);
2         J,I=0;
2         DO WHILE (J < 11) AND (I< VARC);
2              IF (CHAR:=VARC(I:=I+1))='.' THEN J=8;
3              ELSE DO;
3                   BUFFER(J)=CHAR;
4                   J=J+1;
4              END;
3         END;
2         CALL SET$ADDR2(TEMP:=ALLOCATE(164));
2         CALL START$INITIALIZE(TEMP,16);
2         CALL BYTE$OUT(0);
2         CALL STRING$OUT(.BUFFER,11);
2         CALL FILL$STRING(4,0);
2         CALL OR$VALUE(SP-1,1);
2    END BUILD$FCB;
1
1    SET$SIGN: PROCEDURE(NUMB);
2         DECLARE NUMB BYTE;
2         IF GET$TYPE=17 THEN CALL SET$TYPE(VALUE(SP) + NUMB);
2         ELSE CALL PRINT$ERROR('SG');
2         IF VALUE(SP)<>0 THEN CALL SET$S$LENGTH(GET$S$LENGTH + 1);
2    END SET$SIGN;
1
1    PIC$ANALIZER: PROCEDURE;
2         DECLARE     /* WORK AREAS AND VARIABLES */
2         FLAG           BYTE,
2         FIRST          BYTE,
2         COUNT          ADDRESS,
2         BUFFER (31)    BYTE,
2         SAVE           BYTE,
2         REPITITIONS    ADDRESS,
2         J              BYTE,
2         DEC$COUNT      BYTE,
2         CHAR           BYTE,
2         I              BYTE,
2         TEMP           ADDRESS,
2         TYPE           BYTE,
2
2         /* * * MASKS * * */
2         ALPHA     LIT '0',
2         A$EDIT    LIT '2',
2         A$N       LIT '4',
2         EDIT      LIT '8',
2         NUM       LIT '16',
2         NUM$EDIT  LIT '32',
2         DEC       LIT '64',
2         SIGN      LIT '128',
2
2         NUM$MASK          LIT         '10101111B',
2         NUM$ED$MASK       LIT         '10000101B',
2         S$NUM$MASK        LIT         '00101111B',
2         A$E$MASK          LIT         '11111100B',
2         A$N$MASK          LIT         '11101010B',
2         A$N$E$MASK        LIT         '11100000B',
2
2         /* TYPES */
2         NETYPE LIT '80',
2         NTYPE  LIT '16',
2         SNTYPE LIT '17',
2         ATYPE  LIT '8',
2         AETYPE LIT '72',
2         ANTYPE LIT '9',
2         ANETYPE LIT '73';
2
2         INC$COUNT: PROCEDURE(SWITCH);
3              DECLARE SWITCH BYTE;
3              FLAG=FLAG OR SWITCH;
3              IF (COUNT:=COUNT + 1) < 31 THEN BUFFER(COUNT) = CHAR;
3         END INC$COUNT;
2
2         CHECK: PROCEDURE (MASK) BYTE;
3              /* THIS ROUTINE CHECKS A MASK AGINST THE
3              FLAG BYTE AND RETURNS TRUE ID THE FLAG
3              HAC NO BITS IN COMMON WITH THE MASK */
3              DECLARE MASK BYTE;
3              RETURN NOT ( (FLAG AND MASK) <> 0);
3         END CHECK;
2
```

```
2        PIC$ALLOCATE: PROCEDURE(AMT) ADDRESS;
3             DECLARE AMT ADDRESS;
3             IF (MAX$INT$MEM:=MAX$INT$MEM - AMT) < NEXT$AVAILABLE
3                THEN CALL FATAL$ERROR ('MO');
3             RETURN MAX$INT$MEM;
3        END PIC$ALLOCATE;
2
2        /* PROCEDURE EXECUTION STARTS HERE */
2
2        COUNT,FLAG,DEC$COUNT=0;
2        /* CHECK FOR EXCESSIVE LENGTH */
2        IF VARC > 20 THEN
2        DO;
2             CALL PRINT$ERROR('PC');
3             RETURN;
3        END;
2        /* SET FLAG BITS AND COUNT LENGTH */
2        I =1;
2        DO WHILE I<=VARC;
2             IF (CHAR:=VARC(I))='A' THEN CALL INC$COUNT(ALPHA);
3             ELSE IF CHAR ='B' THEN CALL INC$COUNT(A$EDIT);
3             ELSE IF CHAR ='9' THEN CALL INC$COUNT(NUM);
3             ELSE IF CHAR ='X' THEN CALL INC$COUNT(A$N);
3             ELSE IF (CHAR='S') AND (COUNT=0) THEN
3                  FLAG=FLAG OR SIGN;
3             ELSE IF (CHAR = 'V') AND (DEC$COUNT=0) THEN
3                  DEC$COUNT=COUNT;
3             ELSE IF(CHAR='/') OR (CHAR='0') THEN CALL INC$COUNT(EDIT);
3             ELSE IF
3                  (CHAR='Z') OR (CHAR=',') OR (CHAR='*') OR
3                  (CHAR='+') OR (CHAR='-') OR (CHAR='$') THEN
3                  CALL INC$COUNT(NUM$EDIT);
3             ELSE IF (CHAR='.') AND (DEC$COUNT=0) THEN
3             DO;
3                  CALL INC$COUNT(NUM$EDIT);
4                  DEC$COUNT=COUNT;
4             END;
3             ELSE IF ((CHAR='C') AND (VARC(I+1)='R')) OR
3                  ((CHAR='D') AND (VARC(I+1)='B')) THEN
3             DO;
3                  CALL INC$COUNT(NUM$EDIT);
4                  CHAR=VARC(I:=I+1);
4                  CALL INC$COUNT(NUM$EDIT);
4             END;
3             ELSE IF (CHAR='(') AND (COUNT<>0) THEN
3             DO;
3                  SAVE=VARC(I-1);
4                  REPITITIONS=0;
4                  DO WHILE(CHAR:=VARC(I:=I+1))<>')';
4                       REPITITIONS=SHL(REPITITIONS,3) +
5                       SHL(REPITITIONS,1) +(CHAR -'0');
5                  END;
4                  CHAR=SAVE;
4                  DO J=1 TO REPITITIONS-1;
4                       CALL INC$COUNT(0);
5                  END;
4             END;
3             ELSE DO;
3                  CALL PRINT$ERROR('PC');
4                  RETURN;
4             END;
3             I=I+1;
3        END; /* END OF DO WHILE I<= VARC */
2        /* AT THIS POINT THE TYPE CAN BE DETERMINED */
2        IF  NOT CHECK(NUM$EDIT) THEN
2        DO;
2             IF CHECK(NUM$ED$MASK) THEN TYPE=NETYPE;
3        END;
2        ELSE IF CHECK(NUM$MASK) THEN TYPE=NTYPE;
2        ELSE IF CHECK(SNUM$MASK) THEN TYPE=S$N$TYPE;
2        ELSE IF CHECK(NOT(ALPHA)) THEN TYPE=ATYPE;
2        ELSE IF CHECK(A$E$MASK) THEN TYPE =AETYPE;
2        ELSE IF CHECK(A$N$MASK) THEN TYPE=ANTYPE;
2        ELSE IF CHECK(A$N$E$MASK) THEN TYPE=ANETYPE;
2        IF TYPE=0 THEN CALL PRINT$ERROR('PC');
2        ELSE DO;
2             IF REDEF THEN CUR$SYM=REDEF$TWO;
3             ELSE CUR$SYM = HOLD$SYM;
3             CALL SET$TYPE(TYPE);
3             CALL SET$S$LENGTH(COUNT + GET$S$LENGTH);
3             IF (TYPE AND 64) <> 0 THEN
3             DO;
3                  CALL SET$ADDR2(TEMP:=PIC$ALLOCATE(COUNT));
4                  CALL START$INITIALIZE(TEMP,COUNT);
4                  CALL STRING$OUT(.BUFFER + 1,COUNT);
4             END;
3             IF DEC$COUNT<>0 THEN CALL SET$DECIMAL(COUNT-DEC$COUNT);
3        END;
2    END PIC$ANALIZER;
1
1    SET$FILE$ATTRIB: PROCEDURE;
2        DECLARE TEMP ADDRESS, TYPE BYTE;
2        IF CUR$SYM<>VALUE(MPP1) THEN
2        DO;
2             TEMP=CUR$SYM;
3             CUR$SYM=VALUE(MPP1);
3             SYMBOL$ADDR(REL$ID)=TEMP;
3        END;
2        IF NOT (TEMP:=VALUE(SP-1)) THEN CALL PRINT$ERROR ('NF');
2        ELSE  DO;
2             IF TEMP=1 THEN TYPE=SEQUENTIAL;
3             ELSE IF TEMP=15 THEN TYPE=RANDOM;
3             ELSE IF TEMP=9 THEN TYPE=SEQ$RELATIVE;
3             ELSE DO;
3                  CALL PRINT$ERROR('IA');
4                       TYPE=1;
4             END;
3             END;
2        CALL SET$TYPE(TYPE);
2    END SET$FILE$ATTRIB;
```

```
1
1   LOAD$LITERAL: PROCEDURE;
2       DECLARE I BYTE;
2       IF PENDING$LITERAL <> 0 THEN CALL PRINT$ERROR ('LE');
2       ELSE DO I = 0 TO VARC;
3               HOLD$LIT(I)=VARC(I);
3       END;
2   END LOAD$LITERAL;
1
1
1   CHECK$FOR$LEVEL: PROCEDURE;
2        DECLARE NEW$LEVEL BYTE;
2        HOLD$SYM,CUR$SYM=VALUE(MP-1);
2        CALL SET$LEVEL(NEW$LEVEL:=VALUE(MP-2));
2        IF NEW$LEVEL=1 THEN
2        DO;
2             IF ID$STACK<>0 THEN
3             DO;
3                  IF NOT FILE$SEC$END THEN
4                  DO;
4                       CALL SET$REDEF(ID$STACK,VALUE(MP-1));
5                       VALUE(MP)=1;     /* SET REDEFINE FLAG */
5                  END;
4                  CALL END$OF$RECORD;
4             END;
3        END;
2        ELSE DO WHILE STACK$LEVEL >= NEW$LEVEL;
2             CALL REDUCE$STACK;
3        END;
2   END CHECK$FOR$LEVEL;
1
1
1   CODE$GEN: PROCEDURE(PRODUCTION);
2       DECLARE PRODUCTION BYTE;
2       IF PRINT$PROD THEN
2       DO;
2           CALL CRLF;
3           CALL PRINTCHAR(POUND);
3           CALL PRINT$NUMBER(PRODUCTION);
3       END;
2
2       DO CASE PRODUCTION;
2
2     /*  P R O D U C T I O N S */
2
2    /* CASE 0 NOT USED                                                          */
2
3    /*       1    <PROGRAM> ::= <ID-DIV> <E-DIV> <D-DIV> PROCEDURE               */
3       COMPILING=FALSE;
3    /*       2    <ID-DIV> ::= IDENTIFICATION DIVISION . PROGRAM-ID .            */
3    /*       2                  <COMMENT> . <AUTH> <DATE> <SEC>                   */
3      ;      /* NO ACTION REQUIRED */
3    /*       3    <AUTH> ::= AUTHOR . <COMMENT> .                                */
3      ;      /* NO ACTION REQUIRED */
3    /*       4                   <EMPTY>                                         */
3      ;      /* NO ACTION REQUIRED */
3    /*       5    <DATE> ::= DATE-WRITTEN . <COMMENT> .                          */
3      ;      /* NO ACTION REQUIRED */
3    /*       6                   <EMPTY>                                         */
3      ;      /* NO ACTION REQUIRED */
3    /*       7    <SEC> ::= SECURITY . <COMMENT> .                               */
3      ;      /* NO ACTION REQUIRED */
3    /*       8                  <EMPTY>                                          */
3      ;      /* NO ACTION REQUIRED */
3    /*       9    <COMMENT> ::= <INPUT>                                          */
3      ;      /* NO ACTION REQUIRED */
3    /*      10                       <COMMENT> <INPUT>                           */
3      ;      /* NO ACTION REQUIRED */
3    /*      11    <E-DIV> ::= ENVIRONMENT DIVISION . CONFIGURATION               */
3    /*      11                  SECTION . <SRC-OBJ> <I-O>                        */
3      ;      /* NO ACTION REQUIRED */
3    /*      12    <SRC-OBJ> ::= SOURCE-COMPUTER . <COMMENT> <DEBUG> .            */
3    /*      12                  OBJECT-COMPUTER . <COMMENT> .                    */
3      ;      /* NO ACTION REQUIRED */
3    /*      13    <DEBUG> ::= DEBUGGING MODE                                     */
3       DEBUGGING=TRUE;      /* SETS A SCANNER TOGGLE */
3    /*      14                   <EMPTY>                                         */
3      ;      /* NO ACTION REQUIRED */
3    /*      15    <I-O> ::= INPUT-OUTPUT SECTION . FILE-CONTROL .                */
3    /*      15                <FILE-CONTROL-LIST> <IC>                           */
3      ;      /* NO ACTION REQUIRED */
3    /*      16                <EMPTY>                                            */
3      ;      /* NO ACTION REQUIRED */
3    /*      17    <FILE-CONTROL-LIST> ::= <FILE-CONTROL-ENTRY>                   */
3      ;      /* NO ACTION REQUIRED */
3    /*      18                             <FILE-CONTROL-LIST>                   */
3    /*      18                             <FILE-CONTROL-ENTRY>                  */
3      ;      /* NO ACTION REQUIRED */
3    /*      19    <FILE-CONTROL-ENTRY> ::= SELECT <ID> <ATTRIBUTE-LIST> .        */
3       CALL SET$FILE$ATTRIB;
3    /*      20    <ATTRIBUTE-LIST> ::= <ONE-ATTRIB>                              */
3      ;      /* NO ACTION REQUIRED */
3    /*      21                          <ATTRIBUTE-LIST> <ONE-ATTRIB>            */
3       VALUE(MP)=VALUE(SP) OR VALUE(MP);
3    /*      22    <ONE-ATTRIB> ::= ORGANIZATION <ORG-TYPE>                       */
3       VALUE(MP)=VALUE(SP);
3    /*      23                       ACCESS <ACC-TYPE> <RELATIVE>                */
3       VALUE(MP)=VALUE(MPP1) OR VALUE(SP);
3    /*      24                       ASSIGN <INPUT>                              */
3       CALL BUILD$FCB;
3    /*      25    <ORG-TYPE> ::= SEQUENTIAL                                      */
3      ;      /* NO ACTION REQUIRED - DEFAULT */
3    /*      26                     RELATIVE                                      */
3       CALL OR$VALUE(SP,4);
3    /*      27    <ACC-TYPE> ::= SEQUENTIAL                                      */
3      ;      /* NO ACTION REQUIRED - DEFAULT */
3    /*      28                     RANDOM                                        */
3       CALL OR$VALUE(SP,2);
3    /*      29    <RELATIVE> ::= RELATIVE <ID>                                   */
3       CALL OR$VALUE(MP,8);
```

```
3      /*    30                 <EMPTY>                                  */
3       ;     /* NO ACTION REQUIRED - DEFAULT */
3      /*    31   <IC> ::= I-O-CONTROL . <SAME-LIST>                     */
3       ;
3      /*    32             <EMPTY>                                      */
3       ;
3      /*    33   <SAME-LIST> ::= <SAME-ELEMENT>                         */
3       ;
3      /*    34                   <SAME-LIST> <SAME-ELEMENT>             */
3       ;
3      /*    35   <SAME-ELEMENT> ::= SAME <ID-STRING> .                  */
3       ;
3      /*    36   <ID-STRING> ::= <ID>                                   */
3       ;
3      /*    37                   <ID-STRING> <ID>                       */
3       ;
3
3      /*    38   <D-DIV> ::= DATA DIVISION . <FILE-SECTION> <WORK>      */
3      /*    38             <LINK>                                       */
3       ;     /* NO ACTION REQUIRED */
3      /*    39   <FILE-SECTION> ::= FILE SECTION . <FILE-LIST>          */
3       FILE$SEC$END = TRUE;
3      /*    40                     <EMPTY>                              */
3       FILE$SEC$END=TRUE;
3      /*    41   <FILE-LIST> ::= <FILES>                                */
3       ;     /* NO ACTION REQUIRED */
3      /*    42                   <FILE-LIST> <FILES>                    */
3       ;     /* NO ACTION REQUIRED */
3      /*    43   <FILES> ::= FD <ID> <FILE-CONTROL> .                   */
3      /*    43             <RECORD-DESCRIPTION>                         */
3       DO;
3             CALL END$OF$RECORD;
4             CUR$SYM=VALUE(MPP1);
4             CALL SET$ADDRESS(TEMP$HOLD);
4             CALL SET$S$LENGTH(TEMP$TWO);
4       END;
3      /*    44   <FILE-CONTROL> ::= <FILE-LIST>                         */
3       ;     /* NO ACTION REQUIRED */
3      /*    45                     <EMPTY>                              */
3       ;     /* NO ACTION REQUIRED */
3      /*    46   <FILE-LIST> ::= <FILE-ELEMENT>                         */
3       ;     /* NO ACTION REQUIRED */
3      /*    47                   <FILE-LIST> <FILE-ELEMENT>             */
3       ;     /* NO ACTION REQUIRED */
3      /*    48   <FILE-ELEMENT> ::= BLOCK <INTEGER> RECORDS             */
3       ;     /* NO ACTION REQUIRED - FILES NEVER BLOCKED  */
3      /*    49                     RECORD <REC-COUNT>                   */
3       CALL SET$S$LENGTH(VALUE(SP));
3      /*    50                     LABEL RECORDS STANDARD               */
3       ;     /* NO ACTION REQUIRED */
3      /*    51                     LABEL RECORDS OMITTED                */
3       ;     /* NO ACTION REQUIRED */
3      /*    52                     VALUE OF <ID-STRING>                 */
3       ;     /* NO ACTION REQUIRED */
3      /*    53   <REC-COUNT> ::= <INTEGER>                              */
3       ;     /* NO ACTION REQUIRED - VALUE(SP) CORRECT */
3      /*    54                   <INTEGER> TO <INTEGER>                 */
3       DO;
3       VALUE(MP)=VALUE(SP); /* VARIABLE LENGTH */
4       CALL SET$TYPE(4);    /* SET TO VARIABLE */
4       END;
3      /*    55   <WORK> ::= WORKING-STORAGE SECTION .                   */
3      /*    55             <RECORD-DESCRIPTION>                         */
3       ;     /* NO ACTION REQUIRED */
3      /*    56             <EMPTY>                                      */
3       ;     /* NO ACTION REQUIRED */
3      /*    57   <LINK> ::= LINKAGE SECTION . <RECORD-DESCRIPTION>      */
3       CALL PRINT$ERROR('NI'); /* INTER PROG COMM */
3      /*    58             <EMPTY>                                      */
3       ;     /* NO ACTION REQUIRED */
3      /*    59   <RECORD-DESCRIPTION> ::= <LEVEL-ENTRY>                 */
3       ;     /* NO ACTION REQUIRED */
3      /*    60                         <RECORD-DESCRIPTION>             */
3      /*    60                         <LEVEL-ENTRY>                    */
3       ;     /* NO ACTION REQUIRED */
3      /*    61   <LEVEL-ENTRY> ::= <INTEGER> <DATA-ID> <REDEFINES>      */
3      /*    61                   <DATA-TYPE> .                          */
3       DO;
3             CALL LOAD$LEVEL;
4             IF PENDING$LITERAL<>0 THEN PENDING$LIT$ID=ID$STACK$PTR;
4       END;
3      /*    62   <DATA-ID> ::= <ID>                                     */
3       ;     /* NO ACTION REQUIRED */
3      /*    63                 FILLER                                   */
3       DO;
3             CUR$SYM, VALUE(SP)=NEXT$SYM;
4             CALL BUILD$SYMBOL(0);
4       END;
3      /*    64   <REDEFINES> ::= REDEFINES <ID>                         */
3       DO;
3             CALL SET$REDEF(VALUE(SP),VALUE(SP-2));
4             VALUE(MP)=1;     /* SET REDEFINE FLAG ON */
4             CALL CHECK$FOR$LEVEL;
4       END;
3      /*    65                   <EMPTY>                                */
3       CALL CHECK$FOR$LEVEL;
3      /*    66   <DATA-TYPE> ::= <PROP-LIST>                            */
3       ;     /* NO ACTION REQUIRED */
3      /*    67                   <EMPTY>                                */
3       ;     /* NO ACTION REQUIRED */
3      /*    68   <PROP-LIST> ::= <DATA-ELEMENT>                         */
3       ;     /* NO ACTION REQUIRED */
3      /*    69                   <PROP-LIST> <DATA-ELEMENT>             */
3       ;     /* NO ACTION REQUIRED */
3      /*    70   <DATA-ELEMENT> ::= PIC <INPUT>                         */
3       CALL PIC$ANALIZER;
3      /*    71                     USAGE COMP                           */
3       CALL SET$TYPE(COMP);
3      /*    72                     USAGE DISPLAY                        */
3       ;     /* NO ACTION REQUIRED - DEFAULT */
```

```
    /*      73                            SIGN LEADING <SEPARATE>              */
    CALL SET$SIGN(18);
    /*      74                            SIGN TRAILING <SEPARATE>             */
    CALL SET$SIGN(17);
    /*      75                            OCCURS <INTEGER>                     */
    DO;
          CALL CR$TYPE(128);
          CALL SET$OCCURS(VALUE(SP));
    END;
    /*      76                            SYNC <DIRECTION>                     */
    ;     /* NO ACTION REQUIRED - BYTE MACHINE */
    /*      77                            VALUE <LITERAL>                      */
    DO;
          IF NOT FILE$SEC$END THEN
          DO;
                CALL PRINT$ERROR('VE');
                PENDING$LITERAL=0;
          END;
    END;
    /*      78    <DIRECTION> ::= LEFT                                         */
    ;     /* NO ACTION REQUIRED */
    /*      79                      RIGHT                                      */
    ;     /* NO ACTION REQUIRED */
    /*      80                      <EMPTY>                                    */
    ;     /* NO ACTION REQUIRED */
    /*      81    <SEPARATE> ::= SEPARATE                                      */
    VALUE(SP)=2;
    /*      82                     <EMPTY>                                     */
    ;     /* NO ACTION REQUIRED */
    /*      83    <LITERAL> ::= <INPUT>                                        */
    DO;
          CALL LOAD$LITERAL;
          PENDING$LITERAL=1;
    END;
    /*      84                  <LIT>                                          */
    DO;
          CALL LOAD$LITERAL;
          PENDING$LITERAL=2;
    END;
    /*      85                  ZERO                                           */
    PENDING$LITERAL=3;
    /*      86                  SPACE                                          */
    PENDING$LITERAL=4;
    /*      87                  QUOTE                                          */
    PENDING$LITERAL=5;
    /*      88    <INTEGER> ::= <INPUT>                                        */
    CALL CONVERT$INTEGER;
    /*      89    <ID> ::= <INPUT>                                             */
    VALUE(SP)=MATCH;     /* STORE SYMBOL TABLE POINTERS */


    END;   /* END OF CASE STATEMENT */
END CODE$GEN;

GETIN1: PROCEDURE BYTE;
    RETURN INDEX1(STATE);
END GETIN1;

GETIN2: PROCEDURE BYTE;
    RETURN INDEX2(STATE);
END GETIN2;

INCSP: PROCEDURE;
    SP=SP + 1;
    IF SP >= PSTACKSIZE THEN CALL FATAL$ERROR('SO');
    VALUE(SP)=0;     /* CLEAR VALUE STACK */
END INCSP;

LOOKAHEAD: PROCEDURE;
    IF NOLOOK THEN
    DO;
          CALL SCANNER;
          NOLOOK=FALSE;
          IF PRINT$TOKEN THEN
                DO;
                CALL CRLF;
                CALL PRINT$NUMBER(TOKEN);
                CALL PRINT$CHAR(' ');
                CALL PRINT$ACCUM;
          END;
    END;
END LOOKAHEAD;

NO$CONFLICT: PROCEDURE (CSTATE) BYTE;
    DECLARE (CSTATE,I,J,K) BYTE;
    J=INDEX1(CSTATE);
    K=J + INDEX2(CSTATE) - 1;
    DO I=J TO K;
          IF READ1(I)=TOKEN THEN RETURN TRUE;
    END;
RETURN FALSE;
END NO$CONFLICT;

RECOVER: PROCEDURE BYTE;
    DECLARE (TSP, RSTATE) BYTE;
    DO FOREVER;
          TSP=SP;
          DO WHILE TSP <> 255;
                IF NO$CONFLICT(RSTATE:=STATESTACK(TSP)) THEN
                DO;   /* STATE WILL READ TOKEN */
                      IF SP<>TSP THEN SP = TSP - 1;
                      RETURN RSTATE;
                END;
                TSP = TSP - 1;
          END;
          CALL SCANNER;   /* TRY ANOTHER TOKEN */
    END;
END RECOVER;
```

```
1
1    ENDSPASS: PROCEDURE;
2        /* THIS PROCEDURE STORES THE INFORMATION REQUIRED BY PASS2
2        IN LOCATIONS ABOVE THE SYMBOL TABLE. THE FOLLOWING
2        INFORMATION IS STORED:
2              OUTPUT FILE CONTROL BLOCK
2              COMPILER TOGGLES
2              INPUT BUFFER POINTER
2        THE OUTPUT BUFFER IS ALSO FILLED SO THE CURRENT RECORD IS WRITTEN.
2        */
2
2        CALL BYTE$OUT(SCD);
2        CALL ADDR$OUT(NEXT$AVAILABLE);
2        DO WHILE OUTPUT$PTR<>.OUTPUT$BUFF;
2              CALL BYTE$OUT(0FFH);
3        END;
2
2        CALL MOVE(.OUTPUT$FCB,MAX$MEMORY-PASS1$LEN,PASS1$LEN);
2        GO TO MAX$MEMORY;
2    END ENDSPASS;
1
1        /*  *   *   *   *  PROGRAM EXECUTION STARTS HERE *   *   */
1
1    CALL MOVE(INITIAL$POS,MAX$MEMORY,RDR$LENGTH);
1    CALL INIT$SCANNER;
1    CALL INIT$SYMBOL;
1
1
1        /*  *   *   *   *   *   *  PARSER *  *   *   *   *   */
1
1    DO WHILE COMPILING;
1        IF STATE <= MAXRNO THEN          /* READ STATE */
2        DO;
2              CALL INCSP;
3              STATESTACK(SP) = STATE;  /* SAVE CURRENT STATE */
3              CALL LOOKAHEAD;
3              I=GETIN1;
3              J = I + GETIN2 - 1;
3              DO I=I TO J;
3                    IF READ1(I) = TOKEN THEN
4                    DO;
4                    /* COPY THE ACCUMULATOR IF IT IS AN INPUT
4                    STRING.  IF IT IS A RESERVED WORD IT DOES
4                    NOT NEED TO BE COPIED. */
4                          IF (TOKEN=INPUT$STR) OR (TOKEN=LITERAL) THEN
5                                DO K=0 TO ACCUM;
5                                      VARC(K)=ACCUM(K);
6                                END;
5                          STATE=READ2(I);
5                          NOLOCK=TRUE;
5                          I=J;
5                    END;
4                    ELSE
4                    IF I=J THEN
4                    DO;
4                          CALL PRINT$ERROR('NP');
5                          CALL PRINT(.' ERROR NEAR $');
5                          CALL PRINT$ACCUM;
5                          IF (STATE:=RECOVER)=0 THEN COMPILING=FALSE;
5                    END;
4              END;
3        END;     /* END OF READ STATE */
2        ELSE
2        IF STATE>MAXPNO THEN        /* APPLY PRODUCTION STATE */
2        DO;
2              MP=SP - GETIN2;
3              MPP1=MP + 1;
3              CALL CODE$GEN(STATE - MAXPNO);
3              SP=MP;
3              I=GETIN1;
3              J=STATESTACK(SP);
3              DO WHILE (K:=APPLY1(I)) <> 0 AND J<>K;
3                    I=I + 1;
4              END;
3              IF (K:=APPLY2(I))=0 THEN COMPILING=FALSE;
3              STATE=K;
3        END;
2        ELSE
2        IF STATE<=MAXLNO THEN     /*LOOKAHEAD STATE*/
2        DO;
2        I=GETIN1;
3              CALL LOOKAHEAD;
3              DO WHILE (K:=LOOK1(I))<>0 AND TOKEN <>K;
3                    I=I+1;
4              END;
3        STATE=LOOK2(I);
3        END;
2        ELSE
2        DO;          /*PUSH STATES*/
2              CALL INCSP;
3              STATESTACK(SP)=GETIN2;
3              STATE=GETIN1;
3        END;
2    END;  /* OF WHILE COMPILING */
1    CALL CRLF;
1    CALL PRINT(.'END OF PART 1   $');
1    CALL ENDSPASS;
1    EOF
```

```
        /*      COBOL COMPILER - PART 2              */

100H:  /*    LOAD POINT */

        /*     GLOBAL DECLARATIONS AND LITERALS     */

DECLARE LIT LITERALLY 'LITERALLY';
DECLARE
    BDOS              LIT        '5H', /* ENTRY TO OPERATING SYSTEM */
    HASH$TAB$ADDR     LIT        '2500H', /* ADDRESS OF THE BOTTOM OF
                                     THE TABLES  FROM PART1 */
    PASS1$LEN         LIT        '46',
    MAX$MEMORY        LIT        '3200H',
    PASS1$TOP         LIT        '3100H',
    BOOT              LIT        '0',
    CR                LIT        '13',
    LF                LIT        '10',
    QUOTE             LIT        '22H',
    POUND             LIT        '23H',
    TRUE              LIT        '1',
    FALSE             LIT        '0',
    FOREVER           LIT        'WHILE TRUE';
DECLARE MAXRNO LITERALLY '83',/* MAX READ COUNT */
        MAXLNO LITERALLY '106',/* MAX LOOK COUNT */
        MAXPNO LITERALLY '121',/* MAX PUSH COUNT */
        MAXSNO LITERALLY '218',/* MAX STATE COUNT */
        STARTS LITERALLY '1';/* START STATE */
DECLARE READ1 DATA(0,62,5,6,8,13,15,19,21,23,25,30,31,40,41,43,44,48,52
    ,53,57,59,47,27,28,35,36,47,27,47,1,28,58,10,34,45,33,12,27,28,35,36
    ,47,3,1,39,22,47,56,1,55,2,29,42,26,18,32,49,51,63,17,4,37,27,38,47
    ,60,54,1,14,11,7,9,50,5,8,13,15,19,21,23,25,30,40,41,43,44,48,52,53
    ,57,59,50,7,16,1,1,5,8,13,15,19,20,21,23,25,30,40,41,43,44,48,52,53
    ,57,59,47,61,9,47,24,0,0);
DECLARE LOOK1 DATA(0,47,0,39,0,2,0,39,0,1,14,0,47,0,29,42,0,2,0,26,0,7,0
    ,16,0,1,14,0,54,0,54,0,54,0,54,0,11,0,1,14,0,1,0,50,0,47,0,24,0,9,47
    ,0);
DECLARE APPLY1 DATA(0,0,24,0,6,0,0,79,0,0,83,0,13,68,70,76,81,0,0,3,83,0
    ,3,83,0,27,0,0,0,0,59,60,61,0,0,0,0,0,0,71,0,0,0,0,0,0,5,8,9,15,16
    ,46,0,0,2,5,6,8,9,14,15,16,20,23,25,26,28,29,30,31,35,36,42,46,77,78
    ,79,82,0,10,32,39,40,51,54,56,0,5,8,9,15,16,30,46,0,54,0,22,0,0,17
    ,24,66,67,0,0,0,3,83,0);
DECLARE READ2 DATA(0,43,6,7,10,12,84,17,19,20,22,25,26,29,30,31,32,34,35
    ,36,39,40,33,202,206,208,207,86,202,86,122,85,179,195,193,194,136
    ,173,211,206,208,207,210,203,130,28,192,198,87,3,37,4,190,189,23,163
    ,169,167,162,163,16,5,182,202,27,86,41,170,2,13,165,8,175,185,6,10
    ,12,84,17,19,20,22,25,29,30,31,32,34,35,36,39,40,185,9,15,131,132,6
    ,10,12,84,17,18,19,20,22,25,29,30,31,32,34,35,36,39,40,199,42,11,199
    ,21,0,0);
DECLARE LOOK2 DATA(0,14,107,24,108,199,200,38,109,143,143,125,46,110,47
    ,47,111,48,197,49,112,113,51,114,54,115,115,56,58,116,59,117,60,118
    ,61,119,64,120,121,121,66,148,69,71,140,77,123,80,137,129,129,83);
DECLARE APPLY2 DATA(0,0,138,62,78,104,79,128,127,106,75,74,152,151,153
    ,178,150,133,134,105,105,137,103,103,140,183,76,161,50,67,156,154
    ,157,155,149,70,135,63,95,147,68,174,81,160,57,187,82,97,145,98,99
    ,96,176,136,191,44,91,88,91,91,216,91,91,218,180,139,89,125,90,91
    ,158,92,159,144,91,126,126,44,146,45,93,52,53,94,204,204,55,212,196
    ,196,196,196,196,196,196,201,73,72,209,213,172,65,101,214,164,100
    ,141,142,102,102,148);
DECLARE INDEX1 DATA(0,1,116,2,22,116,116,23,116,116,28,30,31,74,116,116
    ,116,23,32,33,116,36,37,116,45,116,116,23,116,116,116,116,28,43,23
    ,116,116,44,45,28,28,46,116,48,49,51,116,52,51,54,55,28,60,61,28,62
    ,63,66,67,67,67,67,68,69,70,71,23,23,74,72,74,92,93,94,95,96,97,116
    ,116,118,120,74,116,2,1,3,5,7,9,12,14,17,19,21,23,25,28,30,32,34,36
    ,38,41,43,45,47,49,217,124,124,177,188,181,205,205,184,171,171,171
    ,171,166,215,1,2,2,4,4,6,6,7,7,9,9,10,10,10,12,12,12,12,12,12,12,12
    ,12,12,12,12,12,12,18,18,18,18,19,19,19,19,22,22,22,25,27,27,27,28
    ,28,29,29,29,30,30,34,34,35,35,36,36,37,37,38,38,39,39,39,40,42,43
    ,43,44,44,45,45,46,46,46,47,47,54,55,80,80,80,88,96,96,98,98,98,100
    ,100,100,101,101,106,106,107,107,108);
DECLARE INDEX2 DATA(0,1,1,20,1,1,1,5,1,1,2,1,1,18,1,1,1,5,1,3,1,1,6,1,1
    ,1,1,5,1,1,1,1,2,1,5,1,1,1,1,2,2,2,1,1,2,1,1,2,1,1,5,2,1,1,2,1,3,1,1
    ,1,1,1,1,1,1,1,5,5,18,2,18,1,1,1,1,1,19,1,2,2,1,13,1,20,2,2,2,2,3,2
    ,3,2,2,2,2,3,2,2,2,2,2,3,2,2,2,2,3,14,24,38,46,47,49,51,54,56,58,59
    ,60,61,64,66,6,1,0,0,1,0,1,2,2,1,2,0,0,2,1,0,2,1,0,2,1,1,3,3,2,3,0,1
    ,2,2,4,2,5,4,4,5,1,1,2,2,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1
    ,0,0,1,2,0,0,0,0,0,0,0,0,0,0,0,0,0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    ,1);

    /* END OF TABLES */
DECLARE
    /* JOINT DECLARATIONS */
    /* THE FOLLOWING ITEMS ARE DECLARED TOGETHER IN THIS
    GROUP IN ORDER TO FACILITATE THEIR BEING PASSED FROM
    THE FIRST PART OF THE COMPILER.
    */
    OUTPUT$FCB        (33) BYTE,
    DEBUGGING         BYTE,
    PRINT$PROD        BYTE,
    PRINT$TOKEN       BYTE,
    LIST$INPUT        BYTE,
    SEQ$NUM           BYTE,
    NEXT$SYM          ADDRESS,
    POINTER           ADDRESS,  /* POINTS TO THE NEXT BYTE TO BE READ */
    NEXT$AVAILABLE ADDRESS,
    MAX$INT$MEM     ADDRESS,

    /*   I O BUFFERS AND GLOBALS */
    IN$ADDR ADDRESS INITIAL (5CH),
    INPUTFCB BASED INADDR (33) BYTE,
    OUTPUT$BUFF       (128)           BYTE,
    OUTPUT$PTR                ADDRESS,
    OUTPUT$END                ADDRESS,
    OUTPUT$CHAR       BASED OUTPUT$PTR BYTE;
```

```
1
1       /* GLOBAL COUNTERS */
1   DECLARE
1       CTR BYTE,
1       A$CTR ADDRESS,
1       BASE ADDRESS,
1       B$BYTE BASED BASE BYTE,
1       B$ADDR BASED BASE ADDRESS;
1
1   MON1: PROCEDURE (F,A);
2       DECLARE F BYTE, A ADDRESS;
2       GO TO BDOS;
2   END MON1;
1
1   MON2: PROCEDURE (F,A) BYTE;
2       DECLARE F BYTE, A ADDRESS;
2       GO TO BDOS;
2   END MON2;
1
1   PRINTCHAR: PROCEDURE (CHAR);
2       DECLARE CHAR BYTE;
2       CALL MON1 (2,CHAR);
2   END PRINTCHAR;
1
1   CRLF: PROCEDURE;
2       CALL PRINTCHAR(CR);
2       CALL PRINTCHAR(LF);
2   END CRLF;
1
1   PRINT: PROCEDURE (A);
2       DECLARE A ADDRESS;
2       CALL MON1 (9,A);
2   END PRINT;
1
1   PRINT$ERROR: PROCEDURE (CODE);
2       DECLARE CODE ADDRESS;
2       CALL CRLF;
2       CALL PRINTCHAR(HIGH(CODE));
2       CALL PRINTCHAR(LOW(CODE));
2   END PRINT$ERROR;
1
1   FATAL$ERROR: PROCEDURE(REASON);
2       DECLARE REASON ADDRESS;
2       CALL PRINT$ERROR(REASON);
2       CALL TIME(10);
2       GO TO BOOT;
2   END FATAL$ERROR;
1
1   CLOSE: PROCEDURE;
2       IF MON2(16,.OUTPUT$FCB)=255 THEN CALL FATAL$ERROR('CL');
2   END CLOSE;
1
1   MORE$INPUT: PROCEDURE BYTE;
2       /* READS THE INPUT FILE AND RETURNS TRUE IF A RECORD
2             WAS READ.  FALSE IMPLIES END OF FILE */
2       DECLARE DCNT BYTE;
2       IF (DCNT:=MON2(20,.INPUT$FCB))>1 THEN CALL FATAL$ERROR('BR');
2       RETURN NOT(DCNT);
2   END MORE$INPUT;
1
1   WRITE$OUTPUT: PROCEDURE (LOCATION);
2       /* WRITES OUT A 128 BYTE BUFFER FROM LOCATION*/
2       DECLARE LOCATION ADDRESS;
2       CALL MON1(26,LOCATION); /* SET DMA */
2       IF MON2(21,.OUTPUT$FCB)<>0 THEN CALL FATAL$ERROR('WR');
2       CALL MON1(26,80H);   /*RESET DMA */
2   END WRITE$OUTPUT;
1
1   MOVE: PROCEDURE(SOURCE, DESTINATION, COUNT);
2       /* MOVES FOR THE NUMBER OF BYTES SPECIFIED BY COUNT */
2       DECLARE (SOURCE,DESTINATION) ADDRESS,
2       (S$BYTE BASED SOURCE, D$BYTE BASED DESTINATION, COUNT) BYTE;
2       DO WHILE (COUNT:=COUNT - 1) <> 255;
2            D$BYTE=S$BYTE;
3            SOURCE=SOURCE +1;
3            DESTINATION = DESTINATION + 1;
3       END;
2   END MOVE;
1
1   FILL: PROCEDURE(ADDR,CHAR,COUNT);
2       /* MOVES CHAR INTO ADDR FOR COUNT BYTES */
2       DECLARE ADDR ADDRESS,
2       (CHAR,COUNT,DEST BASED ADDR) BYTE;
2       DO WHILE (COUNT:=COUNT -1)<>255;
2            DEST=CHAR;
3            ADDR=ADDR + 1;
3       END;
2   END FILL;
1
1       /*   *  *  *  *  *   SCANNER LITS *  *  *  *  */
1   DECLARE
1       LITERAL         LIT        '28',
1       INPUT$STR       LIT        '47',
1       PERIOD          LIT        '1',
1       RPARIN          LIT        '3',
1       LPARIN          LIT        '2',
1       INVALID         LIT        '0';
1
1
1       /*  * * * * SCANNER TABLES * * * *    */
1   DECLARE TOKEN$TABLE DATA
1       /* CONTAINS THE TOKEN NUMBER ONE LESS THAN THE FIRST RESERVED WORD
1       FOR EACH LENGTH OF WORD  */
1       (0,0,3,7,12,28,40,47,55,59,62),
1
```

```
1     TABLE DATA('BY','GO','IF','TO','ADD','END','I-O'
1          ,'NOT','RUN','CALL','ELSE','EXIT','FROM','INTO','LESS','MOVE'
1          ,'NEXT','OPEN','PAGE','READ','SIZE','STOP','THRU','ZERO'
1          ,'AFTER','CLOSE','ENTER','EQUAL','ERROR','INPUT','QUOTE','SPACE'
1          ,'TIMES','UNTIL','USING','WRITE','ACCEPT','BEFORE','DELETE'
1          ,'DIVIDE','OUTPUT','DISPLAY','GREATER'
1          ,'INVALID','NUMERIC','PERFORM','REWRITE','ROUNDED','SECTION'
1          ,'DIVISION','MULTIPLY','SENTENCE','SUBTRACT','ADVANCING',
1          'DEPENDING','PROCEDURE','ALPHABETIC'),
1     OFFSET (11) ADDRESS INITIAL
1          /*  NUMBER OF BYTES TO INDEX INTO THE TABLE FOR EACH LENGTH */
1          (0,0,0,8,23,83,143,173,229,261,288),
1
1     WORD$COUNT DATA
1          /* NUMBER OF WORDS OF EACH SIZE */
1          (0,0,4,5,15,12,5,8,4,3,1),
1
1
1          MAX$ID$LEN        LIT         '12',
1          MAX$LEN         . LIT         '10',
1          ADD$END           DATA ('END. '),
1          LOCKED            BYTE INITIAL (0),
1          HOLD              BYTE,
1          BUFFER$END        ADDRESS     INITIAL    (100H),
1          NEXT              BASED POINTER        BYTE,
1          INBUFF            LIT         '80H',
1          CHAR              BYTE       INITIAL(' '),
1          ACCUM             BYTE,
1          P$ACCUM           (30)        BYTE,
1          DISPLAY           BYTE      INITIAL (0),
1          DISPLAY$REST      (73)        BYTE,
1          TOKEN             BYTE;              /*RETURNED FROM SCANNER */
1
1
1          /*  PROCEDURES USED BY THE SCANNER */
1
1     NEXT$CHAR: PROCEDURE BYTE;
2          IF LOCKED THEN
2          DO;
2            . LOCKED=FALSE;
3              RETURN (CHAR:=HOLD);
3          END;
2          IF (POINTER:=POINTER + 1) >= BUFFER$END THEN
2          DO;
2              IF NOT MORE$INPUT THEN
3               DO;
3                   BUFFER$END=.MEMORY;
4                   POINTER=.ADD$END;
4               END;
3               ELSE POINTER=INBUFF;
3          END;
2          RETURN (CHAR:=NEXT);
2     END NEXT$CHAR;
1
1     GET$CHAR: PROCEDURE;
2          /* THIS PROCEDURE IS CALLED WHEN A NEW CHAR IS NEEDED WITHOUT
2          THE DIRECT RETURN OF THE CHARACTER*/
2          CHAR=NEXT$CHAR;
2     END GET$CHAR;
1
1     DISPLAY$LINE: PROCEDURE;
2          IF NOT LIST$INPUT THEN RETURN;
2          DISPLAY(DISPLAY + 1) = '$';
2          CALL PRINT(.DISPLAY$REST);
2          DISPLAY=0;
2     END DISPLAY$LINE;
1
1     LOAD$DISPLAY: PROCEDURE;
2          IF DISPLAY<72 THEN
2               DISPLAY(DISPLAY:=DISPLAY+1)=CHAR;
2          CALL GET$CHAR;
2     END LOAD$DISPLAY;
1
1     PUT: PROCEDURE;
2          IF ACCUM < 30 THEN
2          ACCUM(ACCUM:=ACCUM+1)=CHAR;
2          CALL LOAD$DISPLAY;
2     END PUT;
1
1     EAT$LINE: PROCEDURE;
2          DO WHILE CHAR<>CR;
2               CALL LOAD$DISPLAY;
3          END;
2     END EAT$LINE;
1
1     GET$NO$BLANK: PROCEDURE;
2          DECLARE (N,I) BYTE;
2          DO FOREVER;
2              IF CHAR = ' ' THEN CALL LOAD$DISPLAY;
3              ELSE
3              IF CHAR=CR THEN
3              DO;
3                   CALL DISPLAY$LINE;
4                   IF SEQ$NUM THEN N=8; ELSE N=2;
4                   DO I = 1 TO N;
4                        CALL LOAD$DISPLAY;
5                   END;
4                   IF CHAR = '*' THEN CALL EAT$LINE;
4              END;
3                   ELSE
3                   IF CHAR = ':' THEN
3                   DO;
3                        IF NOT DEBUGGING THEN CALL EAT$LINE;
4                        ELSE
4                        CALL LOAD$DISPLAY;
4                   END;
3              ELSE
3              RETURN;
3          END;   /* END OF DO FOREVER */
2     END GET$NO$BLANK;
```

```
SPACE: PROCEDURE BYTE;
     RETURN (CHAR=' ') OR (CHAR=CR);
END SPACE;

LEFT$PARIN: PROCEDURE BYTE;
     RETURN CHAR = '(';
END LEFT$PARIN;

RIGHT$PARIN: PROCEDURE BYTE;
     RETURN CHAR = ')';
END RIGHT$PARIN;

DELIMITER: PROCEDURE BYTE;
     /* CHECKS FOR A PERIOD FOLLOWED BY A SPACE OR CR*/
     IF CHAR <> '.' THEN RETURN FALSE;
     HOLD=NEXT$CHAR;
     LOOKED=TRUE;
     IF SPACE THEN
     DO;
          CHAR = '.';
          RETURN TRUE;
     END;
     CHAR='.';
     RETURN FALSE;
END DELIMITER;

END$OF$TOKEN: PROCEDURE BYTE;
     RETURN SPACE OR  DELIMITER OR LEFT$PARIN OR RIGHT$PARIN;
END END$OF$TOKEN;

GET$LITERAL: PROCEDURE BYTE;
     DO FOREVER;
          IF NEXT$CHAR= QUOTE THEN RETURN LITERAL;
          CALL PLT;
     END;
END GET$LITERAL;


LOOK$UP: PROCEDURE BYTE;
     DECLARE POINT ADDRESS,
     (HERE BASED POINT,I) BYTE;

     MATCH: PROCEDURE BYTE;
          DECLARE J BYTE;
          DO J=1 TO ACCUM;
               IF HERE(J - 1) <> ACCUM(J) THEN RETURN FALSE;
          END;
          RETURN TRUE;
     END MATCH;

     POINT=OFFSET(ACCUM)+ .TABLE;
     DO I=1 TO WORD$COUNT(ACCUM);
          IF MATCH THEN RETURN I;
          POINT = POINT + ACCUM;
     END;
     RETURN FALSE;
END LOOK$UP;

RESERVED$WORD: PROCEDURE BYTE;
     /* RETURNS THE TOKEN NUMBER OF A RESERVED WORD IF THE CONTENTS OF
     THE ACCUMULATOR IS A RESERVED WORD, OTHERWISE RETURNS ZERO */
     DECLARE VALUE BYTE;
     DECLARE NUMB BYTE;
     IF ACCUM <= MAX$LEN THEN
     DO;
          IF (NUMB:=TOKEN$TABLE(ACCUM))<>0 THEN
          DO;
               IF (VALUE:=LOOK$UP) <> 0 THEN
                    NUMB=NUMB + VALUE;
               ELSE NUMB=0;
          END;
     END;
     RETURN NUMB;
END RESERVED$WORD;

GET$TOKEN: PROCEDURE BYTE;
     ACCUM=0;
     CALL GET$NO$BLANK;
     IF CHAR=QUOTE THEN RETURN GET$LITERAL;
     IF DELIMITER THEN
     DO;
          CALL FLT;
          RETURN PERIOD;
     END;
     IF LEFT$PARIN THEN
     DO;
          CALL FLT;
          RETURN LPARIN;
     END;
     IF RIGHT$PARIN THEN
     DO;
          CALL FLT;
          RETURN RPARIN;
     END;
     DO FOREVER;
          CALL RLT;
          IF END$OF$TOKEN THEN RETURN INPUT$STR;
     END; /* OF DO FOREVER */
END GET$TOKEN;

     /*    END OF SCANNER ROUTINES  */

     /*    SCANNER EXEC  */

SCANNER: PROCEDURE;
     IF(TOKEN:=GET$TOKEN) = INPUT$STR THEN
          IF (CTR:=RESERVED$WORD) <> 0 THEN TOKEN=CTR;
END SCANNER;


PRINT$ACCUM: PROCEDURE;
     ACCUM(ACCUM+1)='$';
     CALL PRINT(.P$ACCUM);
END PRINT$ACCUM;
```

```
PRINT$NUMBER: PROCEDURE(NUMB);
    DECLARE(NUMB,I,CNT,K) BYTE, J DATA(100,10);
    DO I=0 TO 1;
        CNT=0;
        DO WHILE NUMB >= (K:=J(I));
            NUMB=NUMB - K;
            CNT=CNT + 1;
        END;
        CALL PRINTCHAR('0' + CNT);
    END;
    CALL PRINTCHAR('0' + NUMB);
END PRINT$NUMBER;



    /*  *  *  *  END OF SCANNER PROCEDURES  *  *  *  */


    /*  *  *  * . *  SYMBOL TABLE DECLARATIONS * * *  */

DECLARE

CUR$SYM             ADDRESS,          /*SYMBOL BEING ACCESSED*/
SYMBOL              BASED CUR$SYM  BYTE,
SYMBOL$ADDR         BASED CUR$SYM  ADDRESS,
NEXT$SYM$ENTRY      BASED NEXT$SYM      ADDRESS,
HASH$MASK           LIT         '3FH',
S$TYPE              LIT               '2',
DISPLACEMENT        LIT         '12',
OCCURS              LIT         '11',
P$LENGTH            LIT               '3',
FLD$LENGTH          LIT               '3',
LEVEL               LIT               '10',
REL$ID              LIT               '5',
LOCATION            LIT               '2',
START$NAME          LIT               '11',   /*1 LESS*/
FCB$ADDR            LIT         '4',


      /* * * * * * * SYMBOL TYPE LITERALS * * * * * * */


UNRESOLVED          LIT         '255',
LABEL$TYPE          LIT         '32',
MULT$OCCURS         LIT         '128',
GROUP               LIT         '6',
NON$NUMERIC$LIT     LIT         '7',
ALPHA               LIT         '8',
ALPHA$NUM           LIT         '9',
LIT$SPACE           LIT         '10',
LIT$QUOTE           LIT         '11',
LIT$ZERO            LIT         '12',
NUMERIC$LITERAL     LIT         '15',
NUMERIC             LIT         '16',
COMP                LIT         '21',
A$ED                LIT         '72',
A$N$ED              LIT         '73',
NUM$ED              LIT         '80';


    /*  *  *  *  SYMBOL TABLE ROUTINES  *  *  *  */

SET$ADDRESS: PROCEDURE(ADDR);
DECLARE ADDR ADDRESS;
    SYMBOL$ADDR(LOCATION)=ADDR;
END SET$ADDRESS;

GET$ADDRESS: PROCEDURE ADDRESS;
    RETURN SYMBOL$ADDR(LOCATION);
END GET$ADDRESS;

GET$FCB$ADDR: PROCEDURE ADDRESS;
    RETURN SYMBOL$ADDR(FCB$ADDR);
END GET$FCB$ADDR;

GET$TYPE: PROCEDURE BYTE;
    RETURN SYMBOL(S$TYPE);
END GET$TYPE;

SET$TYPE: PROCEDURE(TYPE);
    DECLARE TYPE BYTE;
    SYMBOL(S$TYPE)=TYPE;
END SET$TYPE;

GET$LENGTH: PROCEDURE ADDRESS;
    RETURN SYMBOL$ADDR(FLD$LENGTH);
END GET$LENGTH;

GET$LEVEL: PROCEDURE BYTE;
    RETURN SHR(SYMBOL(LEVEL),4);
END GET$LEVEL;

GET$DECIMAL: PROCEDURE BYTE;
    RETURN SYMBOL(LEVEL) AND 0FH;
END GET$DECIMAL;

GET$P$LENGTH: PROCEDURE BYTE;
RETURN SYMBOL(P$LENGTH);
END GET$P$LENGTH;

BUILD$SYMBOL: PROCEDURE(LEN);
    DECLARE LEN BYTE, TEMP ADDRESS;
    TEMP=NEXT$SYM;
    IF (NEXT$SYM:=.SYMBOL(LEN:=LEN + DISPLACEMENT))
        > MAX$MEMORY THEN CALL FATAL$ERROR('ST');
    CALL FILL (TEMP,0,LEN);
END BUILD$SYMBOL;
```

```
1     AND$OUT$OCCURS: PROCEDURE (TYPE$IN) BYTE;
2         DECLARE TYPE$IN BYTE;
2         RETURN TYPE$IN  AND 127;
2     END AND$OUT$OCCURS;
1
1
1          /*  *  *  *  PARSER DECLARATIONS  *  *  *  */
1     DECLARE
1     PSTACKSIZE      LIT         '30',       /* SIZE OF PARSE STACKS*/
1     VALUE           (PSTACKSIZE)   ADDRESS,   /* TEMP VALUES */
1     STATESTACK      (PSTACKSIZE)   BYTE,   /* SAVED STATES */
1     VALUE2          (PSTACKSIZE)   ADDRESS,      /* VALUE2  STACK*/
1     VARC                  (100)          BYTE,       /*TEMP CHAR STORE*/
1     ID$STACK        (20)       ADDRESS,
1     ID$PTR          BYTE,
1     MAX$BYTE        BASED          MAX$INT$MEM          BYTE,
1     SUB$IND         BYTE       INITIAL   (0),
1     COND$TYPE       BYTE,
1     HOLD$SECTION    ADDRESS,
1     HOLD$SEC$ADDR   ADDRESS,
1     SECTION$FLAG    BYTE       INITIAL (0),
1     L$ADDR          ADDRESS,
1     L$LENGTH        ADDRESS,
1     L$TYPE          BYTE,
1     L$DEC           BYTE,
1     CON$LENGTH      BYTE,
1     COMPILING       BYTE    INITIAL(TRUE),
1     SP              BYTE    INITIAL (255),
1     MP              BYTE,
1     MPP1            BYTE,
1     NOLOOK          BYTE    INITIAL(FALSE),
1     (I,J,K)         BYTE,        /*INDICIES FOR THE PARSER*/
1     STATE           BYTE    INITIAL(STARTS),
1
1          /* * * * * * * * CODE LITERALS * * * * * * * * * */
1
1
1          /* THE CODE LITERALS ARE BROKEN INTO GROUPS DEPENDING
1          ON THE TOTAL LENGTH OF CODE PRODUCED FOR THAT ACTION */
1
1          /* LENGTH ONE */
1     ADD LIT   '1', /* REGISTER ADDITION */
1     SUB LIT   '2', /* REGISTER SUBTRACTION */
1     MUL LIT   '3', /* REGISTER MULTIPLICATION */
1     DIV LIT   '4', /* REGISTER DIVISION */
1     NEG LIT   '5', /* NOT OPERATOR */
1     STP LIT   '6', /* STOP PROGRAM */
1     STI LIT   '7', /* STORE REGISTER 1 INTO REGISTER 0 */
1
1          /* LENGTH TWO */
1     RND LIT   '8', /* ROUND CONTENTS OF REGISTER 1 */
1
1          /* LENGTH THREE */
1     RET LIT   '9', /* RETURN */
1     CLS LIT   '10',        /* CLOSE */
1     SER LIT   '11',        /* SIZE ERROR */
1     BRN LIT   '12',        /* BRANCH */
1     OPN LIT   '13',        /* OPEN FOR INPUT */
1     OP1 LIT   '14',        /* OPEN FOR OUTPUT */
1     OP2 LIT   '15',        /* OPEN FOR I-O */
1     RGT LIT   '16',        /* REGISTER GREATER THAN */
1     RLT LIT   '17',        /* REGISTER LESS THAN */
1     REQ LIT   '18',        /* REGISTER EQUAL */
1     INV LIT   '19',        /* INVALID FILE ACTION */
1     EOR LIT   '20',        /* END OF FILE REACHED */
1
1          /* LENGTH FOUR */
1     ACC LIT   '21',        /* ACCEPT */
1     DIS LIT   '22',        /* DISPLAY */
1     STD LIT   '23',        /* STOP AND DISPLAY */
1     LDI LIT   '24',        /* LOAD COUNTER IMEDIATE */
1
1          /* LENGTH FIVE */
1     DEC LIT   '25',        /* DECREMENT AND BRANCH IF ZERO */
1     STO LIT   '26',        /* STORE NUMERIC */
1     ST1 LIT   '27',        /* STORE SIGNED NUMERIC TRAILING */
1     ST2 LIT   '28',        /* STORE SIGNED NUMERIC LEADING */
1     ST3 LIT   '29',        /* STORE SEPARATE SIGN LEADING */
1     ST4 LIT   '30',        /* STORE SEPARATE SIGN TRAILING */
1     ST5 LIT   '31',        /* STORE COMPUTATIONAL */
1
1          /* LENGTH SIX */
1     LOD LIT   '32',        /* LOAD NUMERIC LITERAL */
1     LD1 LIT   '33',        /* LOAD NUMERIC */
1     LD2 LIT   '34',        /* LOAD SIGNED NUMERIC TRAILING */
1     LD3 LIT   '35',        /* LOAD SIGNED NUMERIC LEADING */
1     LD4 LIT   '36',        /* LOAD SEPARATE SIGN TRAILING */
1     LD5 LIT   '37',        /* LOAD SEPARATE SIGN LEADING */
1     LD6 LIT   '38',        /* LOAD COMPUTATIONAL */
1
1          /* LENGTH SEVEN */
1     PER LIT   '39',        /* PERFORM */
1     CNU LIT   '40',        /* COMPARE FOR UNSIGNED NUMERIC */
1     CNS LIT   '41',        /* COMPARE FOR SIGNED NUMERIC */
1     CAL LIT   '42',        /* COMPARE FOR ALPHABETIC */
1     RWS LIT   '43',        /* REWRITE SEQUENTIAL */
1     DLS LIT   '44',        /* DELETE SEQUENTIAL */
1     RDF LIT   '45',        /* READ SEQUENTIAL */
1     WTF LIT   '46',        /* WRITE SEQUENTIAL */
1     RVL LIT   '47',        /* READ VARIABLE LENGTH */
1     WVL LIT   '48',        /* WRITE VARIABLE LENGTH */
1
1          /* LENGTH NINE */
1     SCR LIT   '49',        /* SUBSCRIPT COMPUTATION */
1     SGT LIT   '50',        /* STRING GREATER THAN */
1     SLT LIT   '51',        /* STRING LESS THAN */
1     SEQ LIT   '52',        /* STRING EQUAL */
1     MOV LIT   '53',        /* MOVE */
1
```

```
1        /* LENGTH 10 */
1    RRS LIT  '54',      /* READ RELATIVE SEQUENTIAL */
1    WRS LIT  '55',      /* WRITE RELATIVE SEQUENTIAL */
1    RRR LIT  '56',      /* READ RELATIVE RANDOM */
1    WRR LIT  '57',      /* WRITE RELATIVE RANDOM */
1    RWR LIT  '58',      /* REWRITE RELATIVE */
1    DLR LIT  '59',      /* DELETE RELATIVE */
1
1        /* LENGTH ELEVEN */
1    MED LIT  '60',      /* MOVE EDITED */
1
1        /* LENGTH THIRTEEN */
1    MNE LIT  '61',      /* MOVE NUMERIC EDITED */
1
1        /* VARIABLE LENGTH */
1    GDP LIT  '62',      /* GO DEPENDING ON */
1
1        /* BUILD DIRECTING ONLY */
1    INT LIT  '63',      /* INITIALIZE STORAGE */
1    BST LIT  '64',      /* BACK STUFF ADDRESS */
1    TER LIT  '65',      /* TERMINATE BUILD */
1    SCD LIT  '66';      /* SET CODE START */
1
1        /*  *  *  *  PARSER ROUTINES  *  *  *  *  */
1
1    DIGIT: PROCEDURE (CHAR) BYTE;
2        DECLARE CHAR BYTE;
2        RETURN (CHAR<='9') AND (CHAR>='0');
2    END DIGIT;
1
1    LETTER: PROCEDURE BYTE;
2        RETURN (CHAR>='A') AND (CHAR<='Z');
2    END LETTER;
1
1
1    INVALID$TYPE: PROCEDURE;
2        CALL PRINT$ERROR('IT');
2    END INVALID$TYPE;
1
1    BYTE$OUT: PROCEDURE(ONE$BYTE);
2        DECLARE ONE$BYTE BYTE;
2        IF (OUTPUT$PTR:=OUTPUT$PTR + 1) > OUTPUT$END THEN
2        DO;
2              CALL WRITE$OUTPUT(.OUTPUT$BUFF);
3              OUTPUT$PTR=.OUTPUT$BUFF;
3        END;
2        OUTPUT$CHAR=ONE$BYTE;
2    END BYTE$OUT;
1
1    ADDR$OUT: PROCEDURE (ADDR);
2        DECLARE ADDR ADDRESS;
2        CALL BYTE$OUT(LOW(ADDR));
2        CALL BYTE$OUT(HIGH (ADDR));
2    END ADDR$OUT;
1
1    INC$COUNT: PROCEDURE(CNT);
2        DECLARE CNT BYTE;
2        IF(NEXT$AVAILABLE:=NEXT$AVAILABLE + CNT)
2              >MAX$INT$MEM THEN CALL FATAL$ERROR('MO');
2    END INC$COUNT;
1
1
1
1    ONE$ADDR$OPP: PROCEDURE(CODE,ADDR);
2        DECLARE CODE BYTE, ADDR ADDRESS;
2        CALL BYTE$OUT(CODE);
2        CALL ADDR$OUT(ADDR);
2        CALL INC$COUNT(3);
2    END ONE$ADDR$OPP;
1
1    NOT$IMPLIMENTED: PROCEDURE;
2        CALL PRINT$ERROR ('NI');
2    END NOT$IMPLIMENTED;
1
1    MATCH: PROCEDURE ADDRESS;
2        /* CHECKS AN IDENTIFIER TO SEE IF IT IS IN THE SYMBOL
2        TABLE.  IF IT IS PRESENT, CUR$SYM IS SET FOR ACCESS,
2        OTHERWISE THE POINTERS ARE SET FOR ENTRY*/
2        DECLARE (POINT,COLLISION BASED POINT) ADDRESS, (HOLD,I) BYTE;
2        IF VARC>MAX$ID$LEN THEN
2              VARC=MAX$ID$LEN;
2        HOLD=0;
2        DO I=1 TO VARC;
2              HOLD=HOLD+VARC(I);
3        END;
2        POINT=HASH$TAB$ADDR + SHL((HOLD AND HASH$MASK),1);
2        DO FOREVER;
2             IF COLLISION=0 THEN
3             DO;
3                   CUR$SYM,COLLISION=NEXT$SYM;
4                   CALL BUILD$SYMBOL(VARC);
4                   SYMBOL(P$LENGTH)=VARC;
4                   DO I=1 TO VARC;
4                         SYMBOL(START$NAME+I)=VARC(I);
5                   END;
4                   CALL SET$TYPE(UNRESOLVED); /* UNRESOLVED LABEL */
4                   RETURN CUR$SYM;
4             END;
3             ELSE
3             DO;
3                   CUR$SYM=COLLISION;
4                   IF (HOLD:=GET$P$LENGTH)=VARC THEN
4                   DO;
4                        I=1;
5                        DO WHILE SYMBOL(START$NAME + I)= VARC(I);
5                         IF (I:=I+1)>HOLD THEN RETURN(CUR$SYM:=COLLISION);
6                        END;
5                  END;
4             END;
3             POINT=COLLISION;
3        END;
2    END MATCH;
```

```
1    SET$VALUE: PROCEDURE(NUMB);
2        DECLARE NUMB ADDRESS;
2        VALUE(MF)=NUMB;
2    END SET$VALUE;
1
1    SET$VALUE2: PROCEDURE(ADDR);
2        DECLARE ADDR ADDRESS;
2        VALUE2(MF)=ADDR;
2    END SET$VALUE2;
1
1
1    SUB$CNT: PROCEDURE BYTE;
2        IF (SUB$IND:=SUB$IND + 1)>8 THEN
2                SUB$IND=1;
2        RETURN SUB$IND;
2    END SUB$CNT;
1
1
1    CODE$BYTE: PROCEDURE (CODE);
2        DECLARE CODE BYTE;
2        CALL BYTE$OUT(CODE);
2        CALL INC$COUNT(1);
2    END CODE$BYTE;
1
1
1    CODE$ADDRESS: PROCEDURE (CODE);
2        DECLARE CODE ADDRESS;
2        CALL ADDR$OUT(CODE);
2        CALL INC$COUNT(2);
2    END CODE$ADDRESS;
1
1
1    INPUT$NUMERIC: PROCEDURE BYTE;
2        DO CTR=1 TO VARC;
2            IF NOT DIGIT(VARC(CTR)) THEN RETURN FALSE;
3        END;
2        RETURN TRUE;
2    END INPUT$NUMERIC;
1
1
1    CONVERT$INTEGER: PROCEDURE ADDRESS;
2        ACTR=0;
2        DO CTR=1 TO VARC;
2            IF NOT DIGIT(VARC(CTR)) THEN CALL PRINT$ERROR('NN');
3            ACTR=SHL(ACTR,3)+SHL(ACTR,1) + VARC(CTR) - '0';
3        END;
2        RETURN ACTR;
2    END CONVERT$INTEGER;
1
1
1    BACKSTUFF: PROCEDURE (ADD1,ADD2);
2        DECLARE (ADD1,ADD2) ADDRESS;
2        CALL BYTE$OUT(BST);
2        CALL ADDR$OUT(ADD1);
2        CALL ADDR$OUT(ADD2);
2    END BACK$STUFF;
1
1
1    UNRESOLVED$BRANCH: PROCEDURE;
2        CALL SET$VALUE(NEXT$AVAILABLE + 1);
2        CALL ONE$ADDR$OPP(BRN,0);
2        CALL SET$VALUE2(NEXT$AVAILABLE);
2    END UNRESOLVED$BRANCH;
1
1
1    BACK$COND: PROCEDURE;
2        CALL BACKSTUFF(VALUE(SP-1),NEXT$AVAILABLE);
2    END BACK$COND;
1
1
1    SET$BRANCH: PROCEDURE;
2        CALL SET$VALUE(NEXT$AVAILABLE);
2        CALL CODE$ADDRESS(0);
2    END SET$BRANCH;
1
1
1    KEEP$VALUES: PROCEDURE;
2        CALL SET$VALUE(VALUE(SP));
2        CALL SET$VALUE2(VALUE2(SP));
2    END KEEP$VALUES;
1
1
1    STANDARD$ATTRIBUTES: PROCEDURE(TYPE);
2         DECLARE TYPE BYTE;
2         CALL CODE$ADDRESS(GET$FCB$ADDR);
2         CALL CODE$ADDRESS(GET$ADDRESS);
2         CALL CODE$ADDRESS(GET$LENGTH);
2         IF TYPE=0 THEN RETURN;
2         CUR$SYM=SYMBOL$ADDR(REL$ID);
2         CALL CODE$ADDRESS(GET$ADDRESS);
2         CALL CODE$BYTE(GET$LENGTH);
2    END STANDARD$ATTRIBUTES;
1
1
1    READ$WRITE: PROCEDURE(INDEX);
2        DECLARE INDEX BYTE;
2
2        IF (CTR:=GET$TYPE)=1 THEN
2        DO;
2            CALL CODE$BYTE(RDF+INDEX);
3            CALL STANDARD$ATTRIBUTES(0);
3        END;
2        ELSE IF CTR=2 THEN
2        DO;
2            CALL CODE$BYTE(RRS+INDEX);
3            CALL STANDARD$ATTRIBUTES(1);
3        END;
2        ELSE IF CTR=3 THEN
2        DO;
2            CALL CODE$BYTE(RRR+INDEX);
3            CALL STANDARD$ATTRIBUTES(1);
3        END;
```

```
        ELSE IF CTR=4 THEN
        DO;
            CALL CODE$BYTE(RVL+INDEX);
            CALL STANDARD$ATTRIBUTES(0);
        END;
        ELSE CALL PRINT$ERROR('FT');
    END READ$WRITE;


    ARITHMETIC$TYPE: PROCEDURE BYTE;
        IF ((L$TYPE:=AND$OUT$OCCURS(L$TYPE))>=NUMERIC$LITERAL)
            OR (L$TYPE<=COMP) THEN RETURN L$TYPE - NUMERIC$LITERAL;
        CALL INVALID$TYPE;
        RETURN 0;
    END ARITHMETIC$TYPE;


    DEL$RWT: PROCEDURE(FLAG);
        DECLARE FLAG BYTE;
        IF (CTR:=GET$TYPE)=3 THEN
        DO;
            IF FLAG THEN CALL CODE$BYTE(RWR);
            ELSE CALL CODE$BYTE(DLR);
            CALL STANDARD$ATTRIBUTES(1);
            RETURN;
        END;
        IF (CTR=2) AND (NOT FLAG) THEN CALL CODE$BYTE(DLS);
        ELSE IF (CTR<>4) AND FLAG THEN CALL CODE$BYTE(RWS);
        ELSE CALL INVALID$TYPE;
        CALL STANDARD$ATTRIBUTES(0);
    END DEL$RWT;


    ATTRIBUTES: PROCEDURE;
        CALL CODE$ADDRESS(L$ADDR);
        CALL CODE$BYTE(L$LENGTH);
        CALL CODE$BYTE(L$DEC);
    END ATTRIBUTES;


    LOAD$L$ID: PROCEDURE(S$PTR);

        DECLARE S$PTR BYTE;
        IF((A$CTR:=VALUE(S$PTR))<NON$NUMERIC$LIT) OR
            (ACTR=NUMERIC$LITERAL) THEN
        DO;
            L$ADDR=VALUE2(SPTR);
            L$LENGTH=CON$LENGTH;
            L$TYPE=A$CTR;
            RETURN;
        END;
        IF A$CTR<=LIT$ZERO THEN
        DO;
            L$TYPE,L$ADDR=A$CTR;
            L$LENGTH=1;
            RETURN;
        END;
        CUR$SYM=VALUE(S$PTR);
        L$TYPE=GET$TYPE;
        L$LENGTH=GET$LENGTH;
        L$DEC=GET$DECIMAL;
        IF(L$ADDR:=VALUE2(S$PTR))=0 THEN L$ADDR=GET$ADDRESS;
    END LOAD$L$ID;


    LOAD$REG: PROCEDURE(REG$NO,PTR);
        DECLARE (REG$NO,PTR) BYTE;
        CALL LOAD$L$ID(PTR);
        CALL CODE$BYTE(LOD+ARITHMETIC$TYPE);
        CALL ATTRIBUTES;
        CALL CODE$BYTE(REG$NO);
    END LOAD$REG;


    STORE$REG: PROCEDURE(PTR);
        DECLARE PTR BYTE;
        CALL LOAD$L$ID(PTR);
        CALL CODE$BYTE(STO + ARITHMETIC$TYPE -1);
        CALL ATTRIBUTES;
    END STORE$REG;


    STORE$CONSTANT: PROCEDURE ADDRESS;
        IF(MAX$INT$MEM:=MAX$INT$MEM - VARC)<NEXT$AVAILABLE
            THEN CALL FATAL$ERROR('MO');
        CALL BYTE$OUT(INT);
        CALL ADDR$OUT(MAX$INT$MEM);
        CALL ADDR$OUT(CON$LENGTH:=VARC);
        DO CTR = 1 TO CON$LENGTH;
            CALL BYTE$OUT(VARC(CTR));
        END;
        RETURN MAX$INT$MEM;
    END STORE$CONSTANT;


    NUMERIC$LIT: PROCEDURE BYTE;
        DECLARE CHAR BYTE;
        DO CTR=1 TO VARC;
            IF NOT( DIGIT(CHAR:=VARC(CTR))
                OR (CHAR='-') OR (CHAR='+')
                OR (CHAR='.')) THEN RETURN FALSE;
        END;
        RETURN TRUE;
    END NUMERIC$LIT;


    ROUND$STORE: PROCEDURE;
        IF VALUE(SP)<>0 THEN
        DO;
            CALL CODE$BYTE(RND);
            CALL CODE$BYTE(L$DEC);
        END;
        CALL STORE$REG(SP-1);
    END ROUND$STORE;
```

```
ADD$SUB: PROCEDURE (INDEX);
    DECLARE INDEX BYTE;
    CALL LOAD$REG(0,MPP1);
    IF VALUE(SP-3)<>0 THEN
    DO;
        CALL LOAD$REG(1,SP-3);
        CALL CODE$BYTE(ADD);
        CALL CODE$BYTE(STI);
    END;
    CALL LOAD$REG(1,SP-1);
    CALL CODE$BYTE(ADD + INDEX);
    CALL ROUND$STORE;
END ADD$SUB;


MULT$DIV: PROCEDURE(INDEX);
    DECLARE INDEX BYTE;
    CALL LOAD$REG(0,MPP1);
    CALL LOAD$REG(1,SP-1);
    CALL CODE$BYTE(MUL + INDEX);
    CALL ROUND$STORE;
END MULT$DIV;


CHECK$SUBSCRIPT: PROCEDURE;
    CUR$SYM=VALUE(MP);
    IF GET$TYPE<MULT$OCCURS THEN
    DO;
        CALL PRINT$ERROR('IS');
        RETURN;
    END;
    IF INPUT$NUMERIC THEN
    DO;
        CALL SET$VALUE2(GET$ADDRESS + (GET$LENGTH * CONVERT$INTEGER));
        RETURN;
    END;
    CUR$SYM=MATCH;
    IF ((CTR:=GET$TYPE)<NUMERIC) OR (CTR>COMP) THEN
        CALL PRINT$ERROR('TE');
    CALL ONE$ADDR$OPP(SCR,GET$ADDRESS);
    CALL CODE$BYTE(SUBSCNT);
    CALL CODE$BYTE(GET$LENGTH);
    CALL SET$VALUE2(SUB$IND);
END CHECK$SUBSCRIPT;


LOAD$LABEL: PROCEDURE;
    CUR$SYM=VALUE(MP);
    IF (A$CTR:=GET$ADDRESS)<>0 THEN
        CALL BACK$STUFF(A$CTR,VALUE2(MP));
    CALL SET$ADDRESS(VALUE2(MP));
    CALL SET$TYPE(LABEL$TYPE);
    IF (A$CTR:=GET$FCB$ADDR)<>0 THEN
            CALL BACK$STUFF(A$CTR,NEXT$AVAILABLE);
    SYMBOL$ADDR(FCB$ADDR)=NEXT$AVAILABLE;
    CALL ONE$ADDR$OPP(RET,0);
END LOAD$LABEL;


LOAD$SEC$LABEL: PROCEDURE;
    A$CTR=VALUE(MP);
    CALL SET$VALUE(HOLD$SECTION);
    HOLD$SECTION=A$CTR;
    A$CTR=VALUE2(MP);
    CALL SET$VALUE2(HOLD$SEC$ADDR);
    HOLD$SEC$ADDR = A$CTR;
    CALL LOAD$LABEL;
END LOAD$SEC$LABEL;


LABEL$ADDR: PROCEDURE(ADDR,HOLD)ADDRESS;
    DECLARE ADDR ADDRESS;
    DECLARE HOLD BYTE;
    CUR$SYM=ADDR;
    IF(CTR:=GET$TYPE)=LABEL$TYPE THEN
    DO;
        IF HOLD THEN RETURN GET$ADDRESS;
        RETURN GET$FCB$ADDR;
    END;
    IF CTR<>UNRESOLVED THEN CALL INVALID$TYPE;
    IF HOLD THEN
    DO;
        A$CTR=GET$ADDRESS;
        CALL SET$ADDRESS(NEXT$AVAILABLE + 1);
        RETURN A$CTR;
    END;
    A$CTR=GET$FCB$ADDR;
    SYMBOL$ADDR(FCB$ADDR)=NEXT$AVAILABLE + 1;
    RETURN A$CTR;
END LABEL$ADDR;


CODE$FOR$DISPLAY: PROCEDURE (POINT);
    DECLARE POINT BYTE;
    CALL LOAD$LIT(POINT);
    CALL ONE$ADDR$OPP(DIS,L$ADDR);
    CALL CODE$BYTE(L$LENGTH);
END CODE$FOR$DISPLAY;


A$AN$TYPE: PROCEDURE BYTE;
 RETURN (L$TYPE=ALPHA) OR (L$TYPE=ALPHA$NUM);
END A$AN$TYPE;
```

```
1
1    NOT$INTEGER: PROCEDURE BYTE;
2         RETURN L$DEC<>0;
2    END NOT$INTEGER;
1
1
1    NUMERIC$TYPE: PROCEDURE BYTE;
2         RETURN (L$TYPE>=NUMERIC) AND (L$TYPE<=COMP);
2    END NUMERIC$TYPE;
1
1
1    GEN$COMPARE: PROCEDURE;
2         DECLARE (H$TYPE,H$DEC) BYTE,
2                 (H$ADDR,H$LENGTH) ADDRESS;
2
2         CALL LOAD$L$ID(MP);
2         L$TYPE=AND$OUT$OCCURS(L$TYPE);
2         IF COND$TYPE=3 THEN   /* COMPARE FOR NUMERIC */
2         DO;
2              IF A$AN$TYPE OR (L$TYPE>COMP) THEN CALL INVALID$TYPE;
3              IF L$TYPE=NUMERIC THEN CALL CODE$BYTE(CNU);
3              ELSE CALL CODE$BYTE(CNS);
3              CALL CODE$ADDRESS(L$ADDR);
3              CALL CODE$ADDRESS(L$LENGTH);
3              CALL SET$BRANCH;
3           END;
2         ELSE IF COND$TYPE=4 THEN
2         DO;
2              IF NUMERIC$TYPE THEN CALL INVALID$TYPE;
3              CALL CODE$BYTE(CAL);
3              CALL CODE$ADDRESS(L$ADDR);
3              CALL CODE$ADDRESS(L$LENGTH);
3              CALL SET$BRANCH;
3         END;
2         ELSE DO;
2              IF NUMERIC$TYPE THEN CTR=1;
3              ELSE CTR=0;
3              H$TYPE=L$TYPE;
3              H$DEC=L$DEC;
3              H$ADDR=L$ADDR;
3              H$LENGTH=L$LENGTH;
3              CALL LOAD$L$ID(SP);
3              IF NUMERIC$TYPE THEN CTR=CTR+1;
3              IF CTR=2 THEN     /* NUMERIC COMPARE */
3              DO;
3                   CALL LOAD$REG(0,MP);
4                   CALL LOAD$REG(1,SP);
4                   CALL CODE$BYTE(SUB);
4                   CALL CODE$BYTE(RGT + COND$TYPE);
4                   CALL SET$BRANCH;
4              END;
3              ELSE DO;
3                   /* ALPHA NUMERIC COMPARE */
3                   IF (H$DEC<>0) OR (H$TYPE=COMP)
4                        OR (L$DEC<>0) OR (L$TYPE=COMP)
4                        OR (H$LENGTH<>L$LENGTH) THEN CALL INVALID$TYPE;
4                   CALL CODE$BYTE(SGT+COND$TYPE);
4                   CALL CODE$ADDRESS(H$ADDR);
4                   CALL CODE$ADDRESS(L$ADDR);
4                   CALL CODE$ADDRESS(H$LENGTH);
4              END;
3         END;
2    END GEN$COMPARE;
1
1
1    MOVE$TYPE: PROCEDURE BYTE;
2         DECLARE
2         HOLD$TYPE BYTE,
2         ALPHA$NUM$MOVE           LIT '0',
2         A$N$ED$MOVE              LIT '1',
2         NUMERIC$MOVE             LIT '2',
2         N$ED$MOVE                LIT '3';
2
2         L$TYPE=AND$OUT$OCCURS(L$TYPE);
2         IF((HOLD$TYPE:=AND$OUT$OCCURS(GET$TYPE))=GROUP) OR (L$TYPE=GROUP)
2              THEN RETURN ALPHA$NUM$MOVE;
2         IF HOLD$TYPE=ALPHA THEN
2              IF A$AN$TYPE OR (L$TYPE=A$ED) OR (L$TYPE=A$N$ED)
2                   THEN RETURN ALPHA$NUM$MOVE;
2         IF HOLD$TYPE=ALPHA$NUM THEN
2         DO;
2              IF NOT$INTEGER THEN CALL INVALID$TYPE;
3              RETURN ALPHA$NUM$MOVE;
3         END;
2         IF (HOLD$TYPE>=NUMERIC) AND (HOLD$TYPE<=COMP) THEN
2         DO;
2              IF (L$TYPE=ALPHA) OR (L$TYPE>COMP) THEN CALL INVALID$TYPE;
3              RETURN NUMERIC$MOVE;
3         END;
2         IF HOLD$TYPE=A$N$ED THEN
2         DO;
2              IF NOT$INTEGER THEN CALL INVALID$TYPE;
3              RETURN A$N$ED$MOVE;
3         END;
2         IF HOLD$TYPE=A$ED THEN
2              IF A$AN$TYPE OR (L$TYPE>COMP) THEN RETURN A$N$ED$MOVE;
2         IF HOLD$TYPE=NUM$ED THEN
2              IF NUMERIC$TYPE OR (L$TYPE=ALPHA$NUM) THEN
2                   RETURN N$ED$MOVE;
2         CALL INVALID$TYPE;
2         RETURN 0;
2    END MOVE$TYPE;
1
```

```
GEN$MOVE:PROCEDURE;
    DECLARE
    LENGTH1 ADDRESS,
    ADDR1 ADDRESS,
    EXTRA ADDRESS;

    ADD$ADD$LEN: PROCEDURE;
            CALL CODE$ADDRESS(ADDR1);
            CALL CODE$ADDRESS(L$ADDR);
            CALL CODE$ADDRESS(L$LENGTH);
    END ADD$ADD$LEN;

    CODE$FOR$EDIT: PROCEDURE;
            CALL ADD$ADD$LEN;
            CALL CODE$ADDRESS(GET$FCB$ADDR);
            CALL CODE$ADDRESS(LENGTH1);
    END CODE$FOR$EDIT;

    CALL LOAD$L$ID(MPP1);
    CLR$SYM=VALUE(SP);
    IF (ADDR1:=VALUE2(SP))=0 THEN ADDR1=GET$ADDRESS;
    LENGTH1=GET$LENGTH;

    DO CASE MOVE$TYPE;

         /* ALPHA NUMERIC MOVE */

        DO;
            IF LENGTH1>L$LENGTH THEN EXTRA=LENGTH1-L$LENGTH;
            ELSE DO;
                 EXTRA=0;
                 L$LENGTH=LENGTH1;
            END;
            CALL CODE$BYTE(MOV);
            CALL ADD$ADD$LEN;
            CALL CODE$ADDRESS(EXTRA);
        END;

        /* ALPHA NUMERIC EDITED */

        DO;
            CALL CODE$BYTE(MED);
            CALL CODE$FOR$EDIT;
        END;

        /* NUMERIC MOVE */

        DO;
            CALL LOAD$REG(2,MPP1);
            CALL STORE$REG(SP);
        END;

        /* NUMERIC EDITED MOVE */

        DO;
            CALL CODE$BYTE(MNE);
            CALL CODE$FOR$EDIT;
            CALL CODE$BYTE(L$DEC);
            CALL CODE$BYTE(GET$DECIMAL);
        END;
    END;
END GEN$MOVE;


CODE$GEN: PROCEDURE(PRODUCTION);
    DECLARE PRODUCTION BYTE;
    IF PRINT$PROD THEN
    DO;
        CALL CRLF;
        CALL PRINTCHAR(POUND);
        CALL PRINT$NUMBER(PRODUCTION);
    END;

    DO CASE PRODUCTION;

  /*  P R O D U C T I O N S */

    /* CASE 0 NOT USED   */
            ;

 /*       1   <P-DIV> ::= PROCEDURE DIVISION <USING> . <PROC-BODY>      */

    DO;
        COMPILING = FALSE;
        IF SECTION$FLAG THEN CALL LOAD$SEC$LABEL;
    END;

 /*       2   <USING> ::= USING <ID-STRING>                               */

    CALL NOT$IMPLIMENTED;    /* INTER PROG COMM */

 /*       3               <EMPTY>                                         */

    ;     /* NO ACTION REQUIRED */

 /*       4   <ID-STRING> ::= <ID>                                        */
```

```
        ID$STACK(ID$PTR:=0)=VALUE(SP);
/*      5                          <ID-STRING> <ID>                        */
   DO;
        IF(ID$PTR:=ID$PTR+1)=20 THEN
        DO;
             CALL PRINT$ERROR('ID');
             ID$PTR=19;
        END;
        ID$STACK(ID$PTR)=VALUE(SP);
   END;
/*      6    <PROC-BODY> ::= <PARAGRAPH>                                   */
   ;    /* NO ACTION REQUIRED */
/*      7                          <PROC-BODY> <PARAGRAPH>                 */
   ;    /* NO ACTION REQUIRED */
/*      8    <PARAGRAPH> ::= <ID> . <SENTENCE-LIST>                        */
   DO;
        IF SECTION$FLAG=0 THEN SECTION$FLAG=2;
        CALL LOAD$LABEL;
   END;
/*      9                          <ID> SECTION .                          */
   DO;
        IF SECTION$FLAG<>1 THEN
        DO;
             IF SECTION$FLAG=2 THEN CALL PRINT$ERROR('PF');
             SECTION$FLAG=1;
             HOLD$SECTION=VALUE(MP);
             HOLD$SEC$ADDR=VALUE2(MP);
        END;
        ELSE CALL LOAD$SEC$LABEL;
   END;
/*     10    <SENTENCE-LIST> ::= <SENTENCE> .                              */
   ;    /* NO ACTION REQUIRED */
/*     11                              <SENTENCE-LIST> <SENTENCE> .       */
   ;    /* NO ACTION REQUIRED */
/*     12    <SENTENCE> ::= <IMPERATIVE>                                   */
   ;    /* NO ACTION REQUIRED */
/*     13                        <CONDITIONAL>                             */
   ;    /* NO ACTION REQUIRED */
/*     14                        ENTER <ID> <OPT-ID>                       */
   CALL NOT$IMPLIMENTED;    /* LANGUAGE CHANGE */
/*     15    <IMPERATIVE> ::= ACCEPT <SUBID>                               */
   DO;
      CALL LOAD$L$ID(SP);
      CALL ONE$ADDR$OPP(ACC,L$ADDR);
      CALL CODE$BYTE(L$LENGTH);
   END;
/*     16                          <ARITHMETIC>                            */
   ;    /* NO ACTION REQUIRED */
/*     17                          CALL <LIT> <USING>                      */
  CALL NOT$IMPLIMENTED;    /* INTER PROG COMM */
/*     18                          CLOSE <ID>                              */
   CALL ONE$ADDR$OPP(CLS,GET$FCB$ADDR);
/*     19                          <FILE-ACT>                              */
   ;    /* NO ACTION REQUIRED */
/*     20                          DISPLAY <LIT/ID> <OPT-LIT/ID>           */
   DO;
        CALL CODE$FOR$DISPLAY(MPP1);
        IF VALUE(SP)<>0 THEN CALL CODE$FOR$DISPLAY(SP);
   END;
/*     21                          EXIT <PROGRAM-ID>                       */
   ;    /* NO ACTION REQUIRED */
/*     22                          GO <ID>                                 */
   CALL ONE$ADDR$OPP(BRN,LABEL$ADDR(VALUE(SP),1));
```

```
   /*      23                      GO <ID-STRING> DEPENDING <ID>                       */

   DO;
         CALL CODE$BYTE(GDP);
         CALL CODE$BYTE(ID$PTR);
         CUR$SYM=VALUE(SP);
         CALL CODE$BYTE(GET$LENGTH);
         CALL CODE$ADDRESS(GET$ADDRESS);
         DO CTR=0 TO ID$PTR;
              CALL CODE$ADDRESS(LABEL$ADDR(ID$STACK(ID$PTR),1));
         END;
   END;

   /*      24                      MOVE <LIT/ID> TO <SUBID>                            */

   CALL GEN$MOVE;

   /*      25                      OPEN <TYPE-ACTION> <ID>                             */

   CALL ONE$ADDR$OPP(OPN + VALUE(MPP1), GET$FCB$ADDR);

   /*      26                      PERFORM <ID> <THRU> <FINISH>                        */

   DO;
         DECLARE (ADDR2,ADDR3) ADDRESS;
         IF VALUE(SP-1)=0 THEN ADDR2=LABEL$ADDR(VALUE(MPP1),0);
         ELSE ADDR2=LABEL$ADDR(VALUE(SP-1),0);
         IF (ADDR3:=VALUE2(SP))=0 THEN ADDR3=NEXT$AVAILABLE + 7;
         ELSE CALL BACKSTUFF(VALUE(SP),NEXT$AVAILABLE + 7);
         CALL ONE$ADDR$OPP(PER,LABEL$ADDR(VALUE(MPP1),1));
         CALL CODE$ADDRESS(ADDR2);
         CALL CODE$ADDRESS(ADDR3);
   END;

   /*      27                      <READ-ID>                                           */

   CALL NOT$IMPLIMENTED;     /* GRAMMAR ERROR */

   /*      28                      STOP <TERMINATE>                                    */

   DO;
         IF VALUE(SP)=0 THEN CALL CODE$BYTE(STP);
         ELSE CALL ONE$ADDR$OPP(STD,VALUE(SP));
   END;

   /*      29   <CONDITIONAL> ::= <ARITHMETIC> <SIZE-ERROR>                            */
   /*      29                      <IMPERATIVE>                                        */

   CALL BACK$COND;

   /*      30                      <FILE-ACT> <INVALID> <IMPERATIVE>                   */

   CALL BACK$COND;

   /*      31                      IF <CONDITION> <ACTION> ELSE                        */
   /*      31                      <IMPERATIVE>                                        */

   DO;
         CALL BACKSTUFF(VALUE(MPP1),VALUE2(SP-2));
         CALL BACKSTUFF(VALUE(SP-2),NEXT$AVAILABLE);
   END;

   /*      32                      <READ-ID> <SPECIAL> <IMPERATIVE>                    */

   CALL BACK$COND;

   /*      33   <ARITHMETIC> ::= ADD <L/ID> <OPT-L/ID> TO <SUBID>                      */
   /*      33                      <ROUND>                                             */

   CALL ADD$SUB(0);

   /*      34                      DIVIDE <L/ID> INTO <SUBID> <ROUND>                  */

   CALL MULT$DIV(1);

   /*      35                      MULTIPLY <L/ID> BY <SUBID> <ROUND>                  */

   CALL MULT$DIV(0);

   /*      36                      SUBTRACT <L/ID> <OPT-L/ID> FROM                     */
   /*      36                      <SUBID> <ROUND>                                     */

   CALL ADD$SUB(1);

   /*      37   <FILE-ACT> ::= DELETE <ID>                                             */

   CALL DEL$RWT(0);

   /*      38                   REWRITE <ID>                                           */

   CALL DEL$RWT(1);

   /*      39                   WRITE <ID> <SPECIAL-ACT>                               */

   CALL READ$WRITE(1);

   /*      40   <CONDITION> ::= <LIT/ID> <NOT> <COND-TYPE>                             */

   CALL GEN$COMPARE;

   /*      41   <COND-TYPE> ::= NUMERIC                                                */

   COND$TYPE=3;

   /*      42                    ALPHABETIC                                            */

   COND$TYPE=4;

   /*      43                    <COMPARE> <LIT/ID>                                    */

   CALL KEEP$VALUES;
```

```
/*      44    <NOT> ::= NOT                                                    */
   CALL CODE$BYTE(NEG);
/*      45              <EMPTY>                                                */
   ;   /* NO ACTION REQUIRED */
/*      46    <COMPARE> ::= GREATER                                            */
   COND$TYPE=0;
/*      47                  LESS                                               */
   COND$TYPE=1;
/*      48                  EQUAL                                              */
   COND$TYPE=2;
/*      49    <ROUND> ::= ROUNDED                                              */
   CALL SET$VALUE(1);
/*      50                <EMPTY>                                              */
   ;   /* NO ACTION REQUIRED */
/*      51    <TERMINATE> ::= <LITERAL>                                        */
   ;   /* NO ACTION REQUIRED */
/*      52                      RUN                                            */
   ;   /* NO ACTION REQUIRED - VALUE(SP) ALREADY ZERO */
/*      53    <SPECIAL> ::= <INVALID>                                          */
   ;   /* NO ACTION REQUIRED */
/*      54                  END                                                */
   DO;
       CALL SET$VALUE(2);
       CALL CODE$BYTE(EOR);
       CALL SET$BRANCH;
   END;
/*      55    <OPT-ID> ::= <SUBID>                                             */
   ;   /* VALUE AND VALUE2 ALREADY SET */
/*      56                                                                     */
   ;   /* VALUE ALREADY ZERO */
/*      57    <ACTION> ::= <IMPERATIVE>                                        */
   CALL UNRESOLVED$BRANCH;
/*      58                 NEXT SENTENCE                                       */
   CALL UNRESOLVED$BRANCH;
/*      59    <THRU> ::= THRU <ID>                                             */
   CALL KEEP$VALUES;
/*      60                                                                     */
   ;   /* NO ACTION REQUIRED */
/*      61    <FINISH> ::= <L/ID> TIMES                                        */
   DO;
       CALL LOAD$L$ID(MP);
       CALL ONE$ADDR$OPP(LDI,L$ADDR);
       CALL CODE$BYTE(L$LENGTH);
       CALL SET$VALUE2(NEXT$AVAILABLE);
       CALL ONE$ADDR$OPP(DEC,0);
       CALL CODE$ADDRESS(0);
       CALL SET$VALUE(NEXT$AVAILABLE);
   END;
/*      62                 UNTIL <CONDITION>                                   */
   CALL KEEP$VALUES;
/*      63                                                                     */
   ;   /* NO ACTION REQUIRED */
/*      64    <INVALID> ::= INVALID                                            */
   DO;
       CALL SET$VALUE(1);
       CALL CODE$BYTE(INV);
       CALL SET$BRANCH;
   END;
/*      65    <SIZE-ERROR> ::= SIZE ERROR                                      */
   DO;
       CALL CODE$BYTE(SER);
       CALL UNRESOLVED$BRANCH;
   END;
```

```
/*      66   <SPECIAL-ACT> ::= <WHEN> ADVANCING <HOW-MANY>                    */
   CALL NOT$IMPLIMENTED;    /* CARRAGE CONTROL */
/*      67                                                                    */
   ;    /* NO ACTION REQUIRED */
/*      68   <WHEN> ::= BEFORE                                                */
   CALL NOT$IMPLIMENTED;    /* CARRAGE CONTROL */
/*      69              AFTER                                                 */
   CALL NOT$IMPLIMENTED;    /* CARRAGE CONTROL */
/*      70   <HOW-MANY> ::= <INTEGER>                                         */
   CALL NOT$IMPLIMENTED;    /* CARRAGE CONTROL */
/*      71                  PAGE                                              */
   CALL NOT$IMPLIMENTED;    /* CARRAGE CONTROL */
/*      72   <TYPE-ACTION> ::= INPUT                                          */
   ;    /* NO ACTION REQUIRED - VALUE(SP) ALREADY ZERO */
/*      73                     OUTPUT                                         */
   CALL  SET$VALUE(1);
/*      74                     I-O                                            */
   CALL SET$VALUE(2);
/*      75   <SUBID> ::= <SUBSCRIPT>                                          */
   ;    /* VALUE AND VALUE2 ALREADY SET */
/*      76               <ID>                                                 */
   ;    /* NO ACTION REQUIRED */
/*      77   <INTEGER> ::= <INPUT>                                            */
   CALL SET$VALUE(CONVERT$INTEGER);
/*      78   <ID> ::= <INPUT>                                                 */
   DO;
      CALL SET$VALUE(MATCH);
      IF GET$TYPE=UNRESOLVED THEN CALL SET$VALUE2(NEXT$AVAILABLE);
   END;
/*      79   <L/ID> ::= <INPUT>                                               */
   DO;
      IF NUMERIC$LIT THEN
      DO;
         CALL SET$VALUE(NUMERIC$LITERAL);
         CALL SET$VALUE2(STORE$CONSTANT);
      END;
      ELSE CALL SET$VALUE(MATCH);
   END;
/*      80              <SUBSCRIPT>                                           */
   ;    /* NO ACTION REQUIRED */
/*      81              ZERO                                                  */
   CALL SET$VALUE(LIT$ZERO);
/*      82   <SUBSCRIPT> ::= <ID> ( <INPUT> )                                 */
   CALL CHECK$SUBSCRIPT;
/*      83   <OPT-L/ID> ::= <L/ID>                                            */
   ;    /* NO ACTION REQUIRED */
/*      84                  <EMPTY>                                           */
   ;    /* VALUE ALREADY SET */
/*      85   <NN-LIT> ::= <LIT>                                               */
   DO;
      CALL SET$VALUE(NON$NUMERIC$LIT);
      CALL SET$VALUE2(STORE$CONSTANT);
   END;
/*      86                SPACE                                               */
   CALL SET$VALUE(LIT$SPACE);
/*      87                QUOTE                                               */
   CALL SET$VALUE(LIT$QUOTE);
/*      88   <LITERAL> ::= <NN-LIT>                                           */
   ;    /* NO ACTION REQUIRED */
```

```
  /*     89                    <INPUT>                                             */

     DO;
          IF NOT NUMERIC$LIT THEN CALL INVALID$TYPE;
          CALL SET$VALUE(NUMERIC$LITERAL);
          CALL SET$VALUE2(STORE$CONSTANT);
     END;

  /*     90                    ZERO                                                */

     CALL SET$VALUE(LIT$ZERO);

  /*     91   <LIT/ID> ::= <L/ID>                                                  */

     ;    /* NO ACTION REQUIRED */

  /*     92                  <NN-LIT>                                              */

     ;    /* NO ACTION REQUIRED */

  /*     93   <OPT-LIT/ID> ::= <LIT/ID>                                            */

     ;    /* NO ACTION REQUIRED */

  /*     94                        <EMPTY>                                         */

     ;    /* NO ACTION REQUIRED */

  /*     95   <PROGRAM-ID> ::= <ID>                                                */

     CALL NOT$IMPLIMENTED;     /* INTER PROG COMM */

  /*     96                                                                        */
     ;    /* NO ACTION REQUIRED */

  /*     97   <READ-ID> ::= READ <ID>                                              */

     CALL READ$WRITE(0);

     END;    /* END OF CASE STATEMENT */
END CODE$GEN;

GETIN1: PROCEDURE BYTE;
     RETURN INDEX1(STATE);
END GETIN1;

GETIN2: PROCEDURE BYTE;
     RETURN INDEX2(STATE);
END GETIN2;

INCSP: PROCEDURE;
     VALUE(SP:=SP + 1)=0;      /* CLEAR THE STACK WHILE INCREMENTING */
     VALUE2(SP)=0;
     IF SP >= PSTACKSIZE THEN CALL FATAL$ERROR('SO');
END INCSP;

LOOKAHEAD: PROCEDURE;
     IF NOLOOK THEN
     DO;
          CALL SCANNER;
          NOLOOK=FALSE;
          IF PRINT$TOKEN THEN
          DO;
               CALL CRLF;
               CALL PRINT$NUMBER(TOKEN);
               CALL PRINT$CHAR(' ');
               CALL PRINT$ACCUM;
          END;
     END;
END LOOKAHEAD;

NO$CONFLICT: PROCEDURE (CSTATE) BYTE;
     DECLARE (CSTATE,I,J,K) BYTE;
     J=INDEX1(CSTATE);
     K=J + INDEX2(CSTATE) - 1;
     DO I=J TO K;
          IF READ1(I)=TOKEN THEN RETURN TRUE;
     END;
RETURN FALSE;
END NO$CONFLICT;

RECOVER: PROCEDURE BYTE;
     DECLARE TSP BYTE, RSTATE BYTE;
     DO FOREVER;
          TSP=SP;
          DO WHILE TSP <> 255;
               IF NO$CONFLICT(RSTATE:=STATESTACK(TSP)) THEN
               DO;  /* STATE WILL READ TOKEN */
                    IF SP<>TSP THEN SP = TSP - 1;
                    RETURN RSTATE;
               END;
               TSP = TSP - 1;
          END;
          CALL SCANNER;   /* TRY ANOTHER TOKEN */
     END;
END RECOVER;
```

```
1        /*  *  *  *  *  PROGRAM EXECUTION STARTS HERE *  *  */
1
1
1        /* INITIALIZATION */
1
1    TOKEN=62;       /* PRIME THE SCANNER WITH -PROCEDURE- */
1    CALL MOVE(PASS1$TOP-PASS1$LEN,.OUTPUT$FCB,PASS1$LEN);
1        /* THIS SETS
1            OUTPUT FILE CONTROL BLOCK
1            TOGGLES
1            READ POINTER
1            NEXT SYMBOL TABLE POINTER
1        */
1    OUTPUT$END=(OUTPUT$PTR:=.OUTPUT$BUFF-1)+128;
1
1        /*  *  *  *  *  *  *  PARSER *  *  *  *  *  */
1
1    DO WHILE COMPILING;
1        IF STATE <= MAXRNO THEN          /* READ STATE */
2        DO;
2            CALL INCSP;
3            STATESTACK(SP) = STATE;  /* SAVE CURRENT STATE */
3            CALL LOOKAHEAD;
3            I=GETIN1;
3            J = I + GETIN2 - 1;
3            DO I=I TO J;
3                IF READ1(I) = TOKEN THEN
4                DO;
4                /* COPY THE ACCUMULATOR IF IT IS AN INPUT
4                STRING.  IF IT IS A RESERVED WORD IT DOES
4                NOT NEED TO BE COPIED. */
4                    IF (TOKEN=INPUT$STR) OR (TOKEN=LITERAL) THEN
5                        DO K=0 TO ACCUM;
5                            VARC(K)=ACCUM(K);
6                        END;
5                    STATE=READ2(I);
5                    NOLOOK=TRUE;
5                    I=J;
5                END;
4                ELSE
4                IF I=J THEN
4                DO;
4                    CALL PRINT$ERROR('NP');
5                    CALL PRINT(.' ERROR NEAR $');
5                    CALL PRINT$ACCUM;
5                    IF (STATE:=RECOVER)=0 THEN COMPILING=FALSE;
5                END;
4            END;
3        END;    /* END OF READ STATE */
2        ELSE
2        IF STATE>MAXPNO THEN        /* APPLY PRODUCTION STATE */
2        DO;
2            MP=SP - GETIN2;
3            MPP1=MP + 1;
3            CALL CODE$GEN(STATE - MAXPNO);
3            SP=MP;
3            I=GETIN1;
3            J=STATESTACK(SP);
3            DO WHILE (K:=APPLY1(I)) <> 0 AND J<>K;
3                I=I + 1;
4            END;
3            IF (K:=APPLY2(I))=0 THEN COMPILING=FALSE;
3            STATE=K;
3        END;
2        ELSE
2        IF STATE<=MAXLNO THEN      /*LOOKAHEAD STATE*/
2        DO;
2        I=GETIN1;
3            CALL LOOKAHEAD;
3            DO WHILE (K:=LOOK1(I))<>0 AND TOKEN <>K;
3                I=I+1;
4            END;
3        STATE=LOOK2(I);
3        END;
2        ELSE
2        DO;          /*PUSH STATES*/
2            CALL INCSP;
3            STATESTACK(SP)=GETIN2;
3            STATE=GETIN1;
3        END;
2    END;  /* OF WHILE COMPILING */
1    CALL BYTE$OUT(TER);
1    DO WHILE OUTPUT$PTR<>.OUTPUT$BUFF;
1        CALL BYTE$OUT(TER);
2    END;
1    CALL CLOSE;
1    CALL CRLF;
1    CALL PRINT(.'END OF PART 2    $');
1    GO TO BOOT;
1    EOF
```

```
    /*          COBOL INTERPRETER        */

100H:    /* LOAD POINT */

    /* GLOBAL DECLARATIONS AND LITERALS    */

DECLARE

LIT        LITERALLY       'LITERALLY',
BDOS            LIT        '5H',      /* ENTRY TO OPERATING SYSTEM */
BOOT            LIT        '0',
CR              LIT        '13',
LF              LIT        '10',
TRUE            LIT        '1',
FALSE           LIT        '0',
FOREVER         LIT        'WHILE TRUE';

    /* UTILITY VARIABLES */

DECLARE

INDEX           BYTE,
A$CTR           ADDRESS,
CTR             BYTE,
BASE            ADDRESS,
B$BYTE          BASED BASE          BYTE,
B$ADDR          BASED BASE          ADDRESS,
HOLD            ADDRESS,
H$BYTE          BASED HOLD          BYTE,
H$ADDR          BASED HOLD          ADDRESS,


    /* CODE POINTERS */

CODE$START              LIT        '2000H',
PROGRAM$COUNTER         ADDRESS,
C$BYTE                  BASED PROGRAM$COUNTER     BYTE,
C$ADDR                  BASED PROGRAM$COUNTER     ADDRESS;


    /* * * * *   GLOBAL INPUT AND OUTPUT ROUTINES * * * * */


DECLARE
CURRENT$FCB ADDRESS,
START$OFFSET            LIT        '36';

MON1: PROCEDURE (F,A);
    DECLARE F BYTE, A ADDRESS;
    GO TO BDOS;
END MON1;

MON2: PROCEDURE (F,A)BYTE;
    DECLARE F BYTE, A ADDRESS;
    GO TO BDOS;
END MON2;

PRINT$CHAR: PROCEDURE (CHAR);
    DECLARE CHAR BYTE;
    CALL MON1 (2,CHAR);
END PRINT$CHAR;

CRLF: PROCEDURE;
    CALL PRINT$CHAR(CR);
    CALL PRINT$CHAR(LF);
END CRLF;

PRINT: PROCEDURE (A);
    DECLARE A ADDRESS;
    CALL CRLF;
    CALL MON1(9,A);
END PRINT;


READ: PROCEDURE(A);
    DECLARE A ADDRESS;
    CALL MON1(10,A);
END READ;


PRINT$ERROR: PROCEDURE (CODE);
    DECLARE CODE ADDRESS;
    CALL CRLF;
    CALL PRINT$CHAR(HIGH(CODE));
    CALL PRINT$CHAR(LOW(CODE));
END PRINT$ERROR;


FATAL$ERROR: PROCEDURE(CODE);
    DECLARE CODE ADDRESS;
    CALL PRINT$ERROR(CODE);
    CALL TIME(10);
    /* DEBUG
    GO TO BOOT;
    DEBUG */
END FATAL$ERROR;


OPEN: PROCEDURE (ADDR) BYTE;
    DECLARE ADDR ADDRESS;
    RETURN MON2(15,ADDR);
END OPEN;


CLOSE: PROCEDURE (ADDR);
    DECLARE ADDR ADDRESS;
    IF MON2(16,ADDR)<>0 THEN CALL FATAL$ERROR('CL');
END CLOSE;
```

```
1
1    DELETE: PROCEDURE;
2         CALL MON1(19,CURRENT$FCB);
2    END DELETE;
1
1
1    MAKE: PROCEDURE (ADDR);
2         DECLARE ADDR ADDRESS;
2         IF MON2(22,ADDR)<>0 THEN CALL FATAL$ERROR('ME');
2    END MAKE;
1
1
1    SET$DMA: PROCEDURE;
2         CALL MON1(26,CURRENT$FCB+ START$OFFSET);
2    END SET$DMA;
1
1
1    DISK$READ: PROCEDURE BYTE;
2         RETURN MON2(20,CURRENT$FCB);
2    END DISK$READ;
1
1
1    DISK$WRITE: PROCEDURE BYTE;
2         RETURN MON2(21,CURRENT$FCB);
2    END DISK$WRITE;
1
1
1         /* * * * * * * * * * UTILITY PROCEDURES * * * * * * * * * * * * * * * */
1
1
1    DECLARE
1    SUBSCRIPT           (8)        ADDRESS;
1
1
1    RES: PROCEDURE(ADDR) ADDRESS;
2         /* THIS PROCEDURE RESOLVES THE ADDRESS OF A SUBSCRIPTED
2         IDENTIFIER OR A LITERAL CONSTANT */
2
2         DECLARE ADDR ADDRESS;
2         IF ADDR > 32 THEN RETURN ADDR;
2         IF ADDR < 9 THEN RETURN SUBSCRIPT(ADDR);
2         DO CASE ADDR - 9;
2              RETURN .'0';
3              RETURN .' ';
3              RETURN .' ';
3         END;
2         RETURN 0;
2    END RES;
1
1
1    MOVE: PROCEDURE(FROM,DESTINATION,COUNT);
2         DECLARE (FROM,DESTINATION,COUNT) ADDRESS,
2              (F BASED FROM, D BASED DESTINATION) BYTE;
2         DO WHILE (COUNT:=COUNT - 1) <> 0FFFFH;
2              D=F;
3              FROM=FROM + 1;
3              DESTINATION=DESTINATION + 1;
3         END;
2    END MOVE;
1
1
1    FILL: PROCEDURE(DESTINATION,COUNT,CHAR);
2         DECLARE (DESTINATION,COUNT) ADDRESS,
2
2              (CHAR,D BASED DESTINATION) BYTE;
2         DO WHILE (COUNT:=COUNT - 1)<> 0FFFFH;
2              D=CHAR;
3              DESTINATION=DESTINATION + 1;
3         END;
2    END FILL;
1
1
1    CONVERT$TO$HEX: PROCEDURE(POINTER,COUNT) ADDRESS;
2         DECLARE POINTER ADDRESS, COUNT BYTE;
2         A$CTR=0;
2         BASE=POINTER;
2         DO CTR = 0 TO COUNT;
2              A$CTR=SHL(A$CTR,3) + SHL(A$CTR,1) + B$BYTE(CTR) - '0';
3         END;
2         RETURN A$CTR;
2    END CONVERT$TO$HEX;
1
1
1         /* * * * * * * * * * CODE CONTROL PROCEDURES * * * * * * * * * */
1
1    DECLARE
1
1    BRANCH$FLAG          BYTE       INITIAL(TRUE);
1
1    INC$PTR: PROCEDURE (COUNT);
2         DECLARE COUNT BYTE;
2         PROGRAM$COUNTER=PROGRAM$COUNTER + COUNT;
2    END INC$PTR;
1
1
1    GET$OP$CODE: PROCEDURE BYTE;
2         CTR=C$BYTE;
2         CALL INC$PTR(1);
2         RETURN CTR;
2    END GET$OP$CODE;
1
```

```
1
1   CONDITIONAL$BRANCH: PROCEDURE(COUNT);
2        /* THIS PROCEDURE CONTROLS BRANCHING INSTRUCTIONS */
2        DECLARE COUNT BYTE;
2        IF NOT BRANCH$FLAG THEN
2        DO;
2             BRANCH$FLAG=TRUE;
3             PROGRAM$COUNTER=C$ADDR(COUNT);
3        END;
2        ELSE CALL INC$PTR(SHL(COUNT,1)+2);
2   END CONDITIONAL$BRANCH;
1
1
1   INCREMENT$OR$BRANCH: PROCEDURE(MARK);
2        DECLARE MARK BYTE;
2        IF MARK THEN CALL INC$PTR(2);
2        ELSE PROGRAM$COUNTER=C$ADDR;
2   END INCREMENT$OR$BRANCH;
1
1        /* * * * * * * * * * *COMPARISONS * * * * * * * * * * * * * * * */
1
1
1
1
1   CHAR$COMPARE: PROCEDURE BYTE;
2        BASE=C$ADDR;
2        HOLD=C$ADDR(1);
2        DO A$CTR=1 TO C$ADDR(2) - 1;
2             IF B$BYTE(A$CTR) > H$BYTE(A$CTR) THEN RETURN 0;
3             IF B$BYTE(A$CTR) < H$BYTE(A$CTR) THEN RETURN 1;
3        END;
2        RETURN 2;
2   END CHAR$COMPARE;
1
1
1   STRING$COMPARE: PROCEDURE(PIVOT);
2        DECLARE PIVOT BYTE;
2        IF CHAR$COMPARE<>PIVOT THEN BRANCH$FLAG=NOT BRANCH$FLAG;
2        CALL CONDITIONAL$BRANCH(3);
2   END STRING$COMPARE;
1
1
1   NUMERIC: PROCEDURE(CHAR) BYTE;
2        DECLARE CHAR BYTE;
2        RETURN (CHAR >='0') AND (CHAR <='9');
2   END NUMERIC;
1
1
1   LETTER: PROCEDURE(CHAR) BYTE;
2        DECLARE CHAR BYTE;
2        RETURN (CHAR >='A') AND (CHAR <='Z');
2   END LETTER;
1
1
1   SIGN: PROCEDURE(CHAR) BYTE;
2        DECLARE CHAR BYTE;
2        RETURN (CHAR='+') OR (CHAR='-');
2   END SIGN;
1
1
1   COMP$NUM$UNSIGNED: PROCEDURE;
2        BASE=C$ADDR;
2        DO A$CTR=0 TO C$ADDR(2)-1;
2             IF NOT NUMERIC(B$BYTE(A$CTR)) THEN
3             DO;
3                  BRANCH$FLAG=NOT BRANCH$FLAG;
4                  RETURN;
4             END;
3        END;
2        CALL CONDITIONAL$BRANCH(2);
2   END COMP$NUM$UNSIGNED;
1
1
1   COMP$NUM$SIGN: PROCEDURE;
2        BASE=C$ADDR;
2        DO A$CTR=0 TO C$ADDR(2)-1;
2             IF NOT(NUMERIC(CTR:=B$BYTE(A$CTR))
3                  OR SIGN(CTR)) THEN
3             DO;
3                  BRANCH$FLAG=NOT BRANCH$FLAG;
4                  RETURN;
4             END;
3        END;
2        CALL CONDITIONAL$BRANCH(2);
2   END COMP$NUM$SIGN;
1
1
1   COMP$ALPHA: PROCEDURE;
2        BASE=C$ADDR;
2        DO A$CTR=0 TO C$ADDR(2)-1;
2             IF NOT LETTER(B$BYTE(A$CTR)) THEN
3             DO;
3                  BRANCH$FLAG=NOT BRANCH$FLAG;
4                  RETURN;
4             END;
3        END;
2        CALL CONDITIONAL$BRANCH(2);
2   END COMP$ALPHA;
1
1
```

```
     /* * * * * * * * * * *NUMERIC OPERATIONS * * * * * * * * * * * */

DECLARE

(R0,R1,R2)           (10)       BYTE, /* REGISTERS */
(SIGN0,SIGN1,SIGN2)             BYTE,
(DEC$PT0,DEC$PT1,DEC$PT2)       BYTE,
OVERFLOW                BYTE,
B$PTR                   BYTE,
SWITCH                  BYTE,
SIGNIF$NO               BYTE,
ZERO$RESULT             BYTE,
ZONE                    LIT        '10H',
POSITIVE                LIT        '1',
NEGITIVE                LIT        '0';

CHECK$FOR$SIGN: PROCEDURE(CHAR) BYTE;
     DECLARE CHAR BYTE;
     IF NUMERIC(CHAR) THEN RETURN POSITIVE;
     IF NUMERIC(CHAR - ZONE) THEN RETURN NEGITIVE;
     CALL PRINT$ERROR('SI');
     RETURN POSITIVE;
END CHECK$FOR$SIGN;

STORE$IMMEDIATE: PROCEDURE;
     DO CTR=0 TO 9;
          R0(CTR)=R2(CTR);
     END;
     DEC$PT0=DEC$PT2;
     SIGN0=SIGN2;
END STORE$IMMEDIATE;

ONE$LEFT: PROCEDURE;
     DECLARE FLAG BYTE;
     IF ((FLAG:=SHR(B$BYTE,4))=0) OR (FLAG=9) THEN
     DO;
          DO CTR=0 TO 8;
               B$BYTE(CTR)=SHL(B$BYTE(CTR),4) OR SHR(B$BYTE(CTR + 1),4);
          END;
          B$BYTE(9)=SHL(B$BYTE(9),4) OR FLAG;
     END;
     ELSE OVERFLOW=TRUE;
END ONE$LEFT;

ONE$RIGHT: PROCEDURE;
     CTR=10;
     DO INDEX=1 TO 9;
          CTR=CTR-1;
          B$BYTE(CTR)=SHR(B$BYTE(CTR),4) OR SHL(B$BYTE(CTR-1),4);
     END;
     B$BYTE=SHR(B$BYTE,4);
END ONE$RIGHT;

SHIFT$RIGHT: PROCEDURE(COUNT);
     DECLARE COUNT BYTE;
     DO CTR=1 TO COUNT;
          CALL ONE$RIGHT;
     END;
END SHIFT$RIGHT;

SHIFT$LEFT: PROCEDURE (COUNT);
     DECLARE COUNT BYTE;
     OVERFLOW=FALSE;
     DO CTR=1 TO COUNT;
          CALL ONE$LEFT;
          IF OVERFLOW THEN RETURN;
     END;
END SHIFT$LEFT;

ALLIGN: PROCEDURE;
     BASE=.R0;
     IF DEC$PT0 > DEC$PT1 THEN CALL SHIFT$RIGHT(DEC$PT0-DEC$PT1);
     ELSE CALL SHIFT$LEFT(DEC$PT1-DEC$PT0);
END ALLIGN;
```

```
1
1    ADD$R0: PROCEDURE(SECOND, DEST);
2         DECLARE (SECOND, DEST) ADDRESS, (CY,A,B,I) BYTE;
2         HOLD= SECOND;
2         BASE = DEST;
2         CY=0;
2         CTR=9;
2         DO INDEX=1 TO 10;
2              A=R0(CTR);
3              B=B$BYTE(CTR);
3              I=DEC(A+CY);
3              CY=CARRY;
3              I=DEC(I + B);
3              CY=(CY OR CARRY) AND 1;
3              B$BYTE(CTR)=I;
3              CTR=CTR-1;
3         END;
2         IF CY THEN
2         DO;
2              CTR=9;
3              DO INDEX = 1 TO 10;
3                   I=R2(CTR);
4                   I=DEC(I+CY);
4                   CY=CARRY AND 1;
4                   R2(CTR)=I;
4                   CTR=CTR-1;
4              END;
3         END;
2    END ADD$R0;
1
1
1    COMPLIMENT: PROCEDURE(NUMB);
2         DECLARE NUMB BYTE;
2         DO CASE NUMB;
2              HOLD=.R0;
3              HOLD=.R1;
3              HOLD=.R2;
3         END;
2         IF SIGN$O(NUMB) THEN SIGNO(NUMB) = NEGITIVE;
2         ELSE SIGN$O(NUMB)= POSITIVE;
2         DO CTR=0 TO 9;
2              H$BYTE(CTR)=99H - H$BYTE(CTR);
3         END;
2    END COMPLIMENT;
1
1
1    CHECK$RESULT: PROCEDURE;
2         IF SHR(R2,4)=9 THEN CALL COMPLIMENT(2);
2         IF SHR(R2,4)<>0 THEN OVERFLOW=TRUE;
2    END CHECK$RESULT;
1
1
1    CHECK$SIGN: PROCEDURE;
2         IF SIGNO AND SIGN1 THEN
2         DO;
2              SIGN2=POSITIVE;
3              RETURN;
3         END;
2         SIGN2=NEGITIVE;
2         IF NOT SIGNO AND NOT SIGN1 THEN RETURN;
2         IF SIGNO THEN CALL COMPLIMENT(1);
2         ELSE CALL COMPLIMENT(0);
2    END CHECK$SIGN;
1
1
1    LEADING$ZEROES: PROCEDURE (ADDR) BYTE;
2         DECLARE COUNT BYTE, ADDR ADDRESS;
2         COUNT=0;
2         BASE=ADDR;
2         DO CTR=0 TO 9;
2              IF (B$BYTE(CTR) AND 0F0H) <> 0 THEN RETURN COUNT;
3              COUNT=COUNT + 1;
3              IF (B$BYTE(CTR) AND 0FH) <> 0 THEN RETURN COUNT;
3              COUNT=COUNT + 1;
3         END;
2         RETURN COUNT;
2    END LEADING$ZEROES;
1
1
1    CHECK$DECIMAL: PROCEDURE;
2         IF DEC$PT2<>(CTR:=C$BYTE(3)) THEN
2         DO;
2              BASE=.R2;
3              IF DEC$PT2 > CTR THEN CALL SHIFT$RIGHT(DEC$PT2-CTR);
3              ELSE CALL SHIFT$LEFT(CTR-DEC$PT2);
3         END;
2         IF LEADING$ZEROES(.R2) < 19 - C$BYTE(2) THEN OVERFLOW = TRUE;
2    END CHECK$DECIMAL;
1
1
1    ADD: PROCEDURE;
2         OVERFLOW=FALSE;
2         CALL ALLIGN;
2         CALL CHECK$SIGN;
2         CALL ADD$R0(.R1,.R2);
2         CALL CHECK$RESULT;
2    END ADD;
1
1
1    ADD$SERIES: PROCEDURE(COUNT);
2         DECLARE (I,COUNT) BYTE;
2         DO I=1 TO COUNT;
2              CALL ADD$R0(.R2,.R2);
3         END;
2    END ADD$SERIES;
1
```

```
SET$MULT$DIV: PROCEDURE;
    OVERFLOW=FALSE;
    IF (SIGN0 AND SIGN1) OR
        (NOT SIGN0 AND NOT SIGN1) THEN SIGN2=POSITIVE;
    ELSE SIGN2=NEGITIVE;
    CALL FILL(.R2,10,0);
END SET$MULT$DIV;

R1$GREATER: PROCEDURE BYTE;
    DECLARE I BYTE;
    DO CTR=0 TO 9;
        IF R1(CTR)>(I:=99H-R0(CTR)) THEN RETURN TRUE;
        IF R1(CTR)<I THEN RETURN FALSE;
    END;
    RETURN TRUE;
END R1$GREATER;

MULTIPLY: PROCEDURE(VALUE);
    DECLARE VALUE BYTE;
    IF VALUE<>0 THEN CALL ADD$SERIES(VALUE);
    BASE=.R0;
    CALL ONE$LEFT;
END MULTIPLY;

DIVIDE: PROCEDURE;
    DECLARE (I,J,K,LZ0,LZ1) BYTE;
    CALL SET$MULT$DIV;
    IF(LZ0:=LEADING$ZEROES(BASE:=.R0))<>
        (LZ1:=LEADING$ZEROES(.R1)) THEN
    DO;
        IF LZ0>LZ1 THEN
        DO;
            CALL SHIFT$LEFT(I:=LZ0-LZ1);
            DEC$PT0=DEC$PT0 + I;
        END;
        ELSE DO;
            CALL SHIFT$RIGHT(I:=LZ1-LZ0);
            DEC$PT0=DECPT0 -I;
        END;
    END;
    DECPT2= 20 - LZ1 + DECPT0 - DECPT1;
    CALL COMPLIMENT(0);
    DO I=LZ1 TO 19;
        J=0;
        DO WHILE R1$GREATER;
            CALL ADD$R0(.R1,.R1);
            J=J+1;
        END;
        K=SHR(I,1);
        IF I THEN R2(K)=R2(K) OR J;
        ELSE R2(K)=R2(K) OR SHL(J,4);
    END;
END DIVIDE;



LOAD$A$CHAR: PROCEDURE(CHAR);
    DECLARE CHAR BYTE;
    IF (SWITCH:=NOT SWITCH) THEN
        B$BYTE(R$PTR)=B$BYTE(R$PTR) OR SHL(CHAR - 30H,4);
    ELSE B$BYTE(R$PTR:=R$PTR-1)=CHAR - 30H;
END LOAD$A$CHAR;

LOAD$NUMBERS: PROCEDURE(ADDR,CNT);
    DECLARE ADDR ADDRESS, (I,CNT)BYTE;
    HOLD=RES(ADDR);
    CTR=CNT;
    DO INDEX = 1 TO CNT;
        CTR=CTR-1;
        CALL LOAD$A$CHAR(H$BYTE(CTR));
    END;
    CALL INC$PTR(5);
END LOAD$NUMBERS;

SET$LOAD: PROCEDURE (SIGN$IN);
    DECLARE SIGN$IN BYTE;
    DO CASE (CTR:=C$BYTE(4));
        BASE=.R0;
        BASE=.R1;
        BASE=.R2;
    END;
    DEC$PT0(CTR)=C$BYTE(3);
    SIGN0(CTR)=SIGN$IN;
    CALL FILL (BASE,10,0);
    R$PTR=9;
    SWITCH=FALSE;
END SET$LOAD;

LOAD$NUMERIC: PROCEDURE;
    CALL SET$LOAD(1);
    CALL LOAD$NUMBERS(C$ADDR,C$BYTE(2));
END LOAD$NUMERIC;
```

```
LOAD$NUM$LIT: PROCEDURE;
     DECLARE(LIT$SIZE,FLAG) BYTE;

     CHAR$SIGN: PROCEDURE;
          LIT$SIZE=LIT$SIZE - 1;
          HOLD=HOLD + 1;
     END CHAR$SIGN;

     LIT$SIZE=C$BYTE(2);
     HOLD=C$ADDR;
     IF H$BYTE='-' THEN
     DO;
          CALL CHAR$SIGN;
          CALL SET$LOAD(NEGITIVE);
     END;
     ELSE DO;
          IF H$BYTE='+' THEN CALL CHAR$SIGN;
          CALL SET$LOAD(POSITIVE);
     END;
     FLAG=0;
     CTR=LIT$SIZE;
     DO INDEX=1 TO LIT$SIZE;
          CTR=CTR-1;
          IF H$BYTE(CTR)='.' THEN FLAG=LIT$SIZE - (CTR+1);
          ELSE CALL LOAD$A$CHAR(H$BYTE(CTR));
     END;
     DEC$PTO(C$BYTE(4))= FLAG;
     CALL INC$PTR(5);
END LOAD$NUM$LIT;


STORE$ONE: PROCEDURE;
     IF(SWITCH:=NOT SWITCH) THEN
          B$BYTE=SHR(H$BYTE,4) OR '0';
     ELSE DO;
          HOLD=HOLD-1;
          B$BYTE=(H$BYTE AND 0FH) OR '0';
     END;
     BASE=BASE-1;
END STORE$ONE;


STORE$AS$CHAR: PROCEDURE(COUNT);
     DECLARE COUNT BYTE;
     SWITCH=FALSE;
     HOLD=.R2 + 9;
     DO CTR=1 TO COUNT;
          CALL STORE$ONE;
     END;
END STORE$AS$CHAR;


SET$ZONE: PROCEDURE (ADDR);
     DECLARE ADDR ADDRESS;
     IF NOT SIGN2 THEN
     DO;
          BASE=ADDR;
          B$BYTE=B$BYTE OR ZONE;
     END;
     CALL INC$PTR(4);
END SET$ZONE;


SET$SIGN$SEP: PROCEDURE (ADDR);
     DECLARE ADDR ADDRESS;
     BASE=ADDR;
     IF SIGN2 THEN B$BYTE='+';
     ELSE B$BYTE='-';
     CALL INC$PTR(4);
END SET$SIGN$SEP;


STORE$NUMERIC: PROCEDURE;
     CALL CHECK$DECIMAL;
     BASE=C$ADDR + C$BYTE(2) -1;
     CALL STORE$AS$CHAR(C$BYTE(2));
END STORE$NUMERIC;




     /* * * * * * * * * INPUT-OUTPUT ACTIONS * * * * * * * * * * * * */


DECLARE

FLAG$OFFSET         LIT         '33',
EXTENT$OFFSET       LIT         '12',
REC$NO              LIT         '32',
PTR$OFFSET          LIT         '17',
BUFF$LENGTH         LIT         '128',
VAR$END             LIT         'CR',
TERMINATOR          LIT              '1AH',
END$OF$RECORD       BYTE,
INVALID             BYTE,
RANDOM$FILE         BYTE,
CURRENT$FLAG        BYTE,
FCB$BYTE            BASED CURRENT$FCB          BYTE,
FCB$ADDR            BASED CURRENT$FCB          ADDRESS,
BUFF$PTR            ADDRESS,
BUFF$END            ADDRESS,
BUFFSTART           ADDRESS,
BUFF$BYTE           BASED BUFF$PTR       BYTE,
CON$BUFF            ADDRESS    INITIAL (80H),
CON$BYTE            BASED CON$BUFF        BYTE,
CON$INPUT           ADDRESS    INITIAL (82H);
```

```
ACCEPT: PROCEDURE;
     CALL CRLF;
     CALL PRINT$CHAR(3FH);
     CALL CRLF;
     CALL FILL(CON$INPUT,(CON$BYTE:=C$BYTE(2)),' ');
     CALL READ(CON$BUFF);
     CALL MOVE(CON$INPUT,RES(C$ADDR),CON$BYTE);
     CALL INC$PTR(3);
END ACCEPT;


DISPLAY: PROCEDURE;
     BASE=C$ADDR;
     CALL CRLF;
     DO CTR= 0 TO C$BYTE(2) - 1;
           CALL PRINT$CHAR(B$BYTE(CTR));
     END;
     CALL INC$PTR(3);
END DISPLAY;


SET$FILE$TYPE: PROCEDURE(TYPE);
     DECLARE TYPE BYTE;
     BASE=C$ADDR;
     B$BYTE(FLAG$OFFSET)=TYPE;
END SET$FILE$TYPE;


GET$FILE$TYPE: PROCEDURE BYTE;
     BASE=C$ADDR;
     RETURN B$BYTE(FLAG$OFFSET);
END GET$FILE$TYPE;


SET$I$O: PROCEDURE;
     END$OF$RECORD,INVALID=FALSE;
     IF C$ADDR=CURRENT$FCB THEN RETURN;
     /* STORE CURRENT POINTERS AND SET INTERNAL WRITE MARK */
     BASE=CURRENT$FCB;
     FCB$ADDR(PTR$OFFSET)=BUFF$PTR;
     FCB$BYTE(FLAG$OFFSET)=CURRENT$FLAG;
     /* LOAD NEW VALUES */
     BUFF$END=(BUFF$START:=(CURRENT$FCB:=C$ADDR)+START$OFFSET)
          + BUFF$LENGTH;
     CURRENT$FLAG=FCB$BYTE(FLAG$OFFSET);
     BUFF$PTR=FCB$ADDR(PTR$OFFSET);
END SET$I$O;


OPEN$FILE: PROCEDURE(TYPE);
     DECLARE TYPE BYTE;
     CALL SET$FILE$TYPE(TYPE);
     CTR=OPEN(CURRENT$FCB:=C$ADDR);
     DO CASE TYPE-1;
          /* INPUT */
          DO;
               IF CTR=255 THEN CALL PRINT$ERROR('NF');
               FCB$ADDR(PTR$OFFSET)=CURRENT$FCB+100H;
          END;
          /* OUTPUT */
          DO;
               CALL DELETE;
               CALL MAKE(C$ADDR);
               FCB$ADDR(PTR$OFFSET)=CURRENT$FCB+START$OFFSET-1;
          END;
          /* I-O */
          DO;
               IF CTR=255 THEN CALL FATAL$ERROR('NF');
               FCB$ADDR(PTR$OFFSET)=CURRENT$FCB + 100H;
          END;
     END;
     CURRENT$FCB=0;     /* FORCE A PARAMETER LOAD */
     CALL SET$I$O;
     CALL INC$PTR(2);
END OPEN$FILE;


WRITE$MARK: PROCEDURE BYTE;
     RETURN ROL(CURRENT$FLAG,1);
END WRITE$MARK;


SET$WRITE$MARK: PROCEDURE;
     CURRENT$FLAG=CURRENT$FLAG OR 80H;
END SET$WRITE$MARK;


WRITE$RECORD: PROCEDURE;
     IF NOT SHR(CURRENT$FLAG,1) THEN CALL FATAL$ERROR('WI');
     CALL SET$DMA;
     CURRENT$FLAG=CURRENT$FLAG AND 0FH;
     IF (CTR:=DISK$WRITE) =0 THEN RETURN;
     INVALID=TRUE;
END WRITE$RECORD;


READ$RECORD: PROCEDURE;
     CALL SET$DMA;
     IF WRITE$MARK THEN CALL WRITE$RECORD;
     IF (CTR:=DISK$READ)=0 THEN RETURN;
     IF CTR=1 THEN END$OF$RECORD=TRUE;
     ELSE INVALID=TRUE;
END READ$RECORD;
```

```
READ$BYTE: PROCEDURE BYTE;
     IF (BUFF$PTR:=BUFF$PTR + 1) >= BUFFEND THEN
     DO;
          CALL READ$RECORD;
          IF END$OF$RECORD THEN RETURN TERMINATOR;
          BUFF$PTR=BUFF$START;
     END;
     RETURN BUFF$BYTE;
END READ$BYTE;


WRITE$BYTE: PROCEDURE (CHAR);
     DECLARE CHAR BYTE;
     IF (BUFF$PTR:=BUFF$PTR+1) >= BUFF$END THEN
     DO;
          CALL WRITE$RECORD;
          BUFF$PTR=BUFF$START;
     END;
     CALL SET$WRITE$MARK;
     BUFF$BYTE=CHAR;
END WRITE$BYTE;


WRITE$END$MARK: PROCEDURE;
     CALL WRITE$BYTE(CR);
     CALL WRITE$BYTE(LF);
END WRITE$END$MARK;


READ$END$MARK: PROCEDURE;
     IF READ$BYTE<>CR THEN CALL PRINT$ERROR('EM');
     IF READ$BYTE<>LF THEN CALL PRINT$ERROR('EM');
END READ$END$MARK;


READ$VARIABLE:PROCEDURE;
     CALL SET$I$C;
     BASE=C$ADDR(1);
     DO A$CTR=0 TO C$ADDR(2)-1;
          IF (CTR:=(B$BYTE(A$CTR):=READ$BYTE)) = VAR$END THEN
          DO;
               CTR=READ$BYTE;
               RETURN;
          END;
          IF CTR=TERMINATOR THEN
          DO;
               END$OF$RECORD=TRUE;
               RETURN;
          END;
     END;
     CALL READ$END$MARK;
END READ$VARIABLE;


WRITE$VARIABLE: PROCEDURE;
     DECLARE COUNT ADDRESS;
     CALL SET$I$C;
     BASE=C$ADDR(1);
     COUNT=C$ADDR(2);
     DO WHILE(B$BYTE(COUNT:=COUNT-1)<>' ')AND (COUNT<>0);
     END;
     DO A$CTR=0 TO COUNT;
          CALL WRITE$BYTE(B$BYTE(A$CTR));
     END;
     CALL WRITE$END$MARK;
END WRITE$VARIABLE;


READ$TO$MEMORY: PROCEDURE;
     CALL SET$I$C;
     BASE=C$ADDR(1);
     DO A$CTR=0 TO C$ADDR(2)-1;
          IF (B$BYTE(A$CTR):=READ$BYTE)=TERMINATOR THEN
          DO;
               END$OF$RECORD=TRUE;
               RETURN;
          END;
     END;
     CALL READ$END$MARK;
END READ$TO$MEMORY;


WRITE$FROM$MEMORY: PROCEDURE;
     CALL SET$I$C;
     BASE=C$ADDR(1);
     DO A$CTR=0 TO C$ADDR(2)-1;
          CALL WRITE$BYTE(B$BYTE(A$CTR));
     END;
     CALL WRITE$END$MARK;
END WRITE$FROM$MEMORY;
```

```
1
1        /* * * * * * * * * RANDOM I-O PROCEDURES * * * * * * * */
1
1
1   SET$RANDOM$POINTER: PROCEDURE;
2       /*
2       THIS PROCEDURE READS THE RANDOM KEY AND COMPUTES
2       WHICH RECORD NEEDS TO BE AVAILABLE IN THE BUFFER
2       THAT RECORD IS MADE AVAILABLE AND THE POINTERS
2       SET FOR INPUT OR OUTPUT
2       */
2       DECLARE (BYTE$COUNT,RECORD) ADDRESS,
2             EXTENT BYTE;
2       CALL SET$I$C;
2       BYTE$COUNT=(C$ADDR(2)+2)*CONVERT$TO$HEX(C$ADDR(3),C$BYTE(8));
2       RECORD=SHR(BYTE$COUNT,7);
2       EXTENT=SHR(RECORD,7);
2       IF EXTENT<>FCB$BYTE(EXTENT$OFFSET) THEN
2       DO;
2       IF WRITE$MARK THEN CALL WRITE$RECORD;
3       CALL CLOSE(C$ADDR);
3             FCB$BYTE(EXTENT$OFFSET)=EXTENT;
3             IF OPEN(C$ADDR)<>0 THEN
3             DO;
3                   IF SHR(CURRENT$FLAG,1) THEN CALL MAKE(C$ADDR);
4                   ELSE INVALID=TRUE;
4             END;
3       END;
2       BUFF$PTR=(BYTE$COUNT AND 7FH) + BUFF$START -1;
2       IF FCB$BYTE(REC$NO)<>(CTR:=LOW(RECORD)AND 7FH) THEN
2       DO;
2       FCB$BYTE(32)=CTR;
3             CALL READ$RECORD;
3       END;
2   END SET$RANDOM$POINTER;
1
1
1   GET$REC$NUMBER: PROCEDURE;
2       DECLARE (RECNUM, K) ADDRESS,
2             (I,CNT) BYTE,
2             J(4) ADDRESS INITIAL (10000,1000,100,10),
2             BUFF(5) BYTE;
2
2       REC$NUM=SHL(FCB$BYTE(EXTENT$OFFSET),7)+FCB$BYTE(REC$NO);
2       DO I=0 TO 3;
2       CNT=0;
3             DO WHILE REC$NUM>=(K:=J(I));
3                   REC$NUM=REC$NUM - K;
4                   CNT=CNT + 1;
4             END;
3             BUFF(I)=CNT + '0';
3       END;
2       BUFF(4)=REC$NUM+'0';
2       IF (I:=C$BYTE(8))<=5 THEN
2       CALL MOVE(.BUFF+4-I,C$ADDR(3),I);
2       ELSE DO;
2       CALL FILL(C$ADDR,I-5,' ');
3             CALL MOVE(.BUFF,C$ADDR(3)+I-6, 5);
3       END;
2   END GET$REC$NUMBER;
1
1
1   WRITE$ZERO$RECORD: PROCEDURE;
2       DO A$CTR=1 TO C$ADDR(2);
2             CALL WRITE$BYTE(0);
3       END;
2   END WRITE$ZERO$RECORD;
1
1
1   WRITE$RANDOM: PROCEDURE;
2       CALL SET$RANDOM$POINTER;
2       CALL WRITE$FROM$MEMORY;
2       CALL INC$PTR(5);
2   END WRITE$RANDOM;
1
1
1   BACK$ONE$RECORD: PROCEDURE;
2       CALL SET$I$C;
2   IF (BUFF$PTR:=BUFF$PTR-(C$ADDR(2)+2))>=BUFF$START THEN RETURN;
2       BUFF$PTR=BUFF$END-(BUFF$START - BUFF$PTR);
2       IF (FCB$BYTE(REC$NO):=FCB$BYTE(REC$NO)-1)=255 THEN
2       DO;
2       FCB$BYTE(EXTENT$OFFSET)=FCB$BYTE(EXTENT$OFFSET)-1;
3             IF OPEN(C$ADDR)<> 0 THEN
3             DO;
3             CALL PRINT$ERROR('OP');
4                   INVALID=TRUE;
4             END;
3             FCB$BYTE(REC$NO)=127;
3       END;
2       CALL READ$RECORD;
2   END BACK$ONE$RECORD;
1
1
1        /* * * * * * * * * * * * * * MOVES * * * * * * * * * * * * * * */
1
1
1   INC$HOLD: PROCEDURE;
2       HOLD=HOLD + 1;
2       CTR=CTR + 1;
2   END INC$HOLD;
1
1
1   LOAD$INC: PROCEDURE;
2       H$BYTE=B$BYTE;
2       BASE=BASE+1;
2       CALL INC$HOLD;
2   END LOAD$INC;
1
```

```
CHECK$EDIT: PROCEDURE(CHAR);
    DECLARE CHAR BYTE;
    IF (CHAR='0') OR (CHAR='/') THEN CALL INC$HOLD;
    ELSE IF CHAR='B' THEN
    DO;
         H$BYTE=' ';
         CALL INC$HOLD;
    END;
    ELSE IF CHAR='A' THEN
    DO;
         IF NOT LETTER(B$BYTE) THEN CALL PRINT$ERROR('IC');
         CALL LOAD$INC;
    END;
    ELSE IF CHAR='9' THEN
    DO;
         IF NOT NUMERIC (B$BYTE) THEN CALL PRINT$ERROR('IC');
         CALL LOAD$INC;
    END;
    ELSE CALL LOAD$INC;
END CHECK$EDIT;

    /* * * * * * * * * * * * MACHINE ACTIONS * *  * * * * * * * * * */


STOP: PROCEDURE;
    CALL PRINT(.'EOF    $');
    GO TO BOOT;
END STOP;


    /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

              THE PROCEDURE BELOW CONTROLS THE EXECUTION OF THE CODE.
              IT DECODES EACH OP-CODE AND PERFORMS THE ACTIONS

    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */


EXECUTE: PROCEDURE;
    DO FOREVER;
         DO CASE GET$OP$CODE;

              ;      /* CASE ZERO NOT USED */

  /* ADD */

              CALL ADD;

  /* SUB */

              DO;
                   CALL COMPLIMENT(0);
                   IF SIGNO THEN SIGNO=NEGITIVE;
                   ELSE SIGNO=POSITIVE;
                   CALL ADD;
              END;

  /* MUL */

              DO;
                   DECLARE I BYTE;
                   CALL SET$MULT$DIV;
                   DECPT1,DECPT2=DECPT1 + DECPT0;
                   CALL ALLIGN;
                   CALL MULTIPLY(SHR(R1(I:=9),4));
                   DO INDEX=1 TO 9;
                        CALL MULTIPLY(R1(I:=I-1) AND 0FH);
                        CALL MULTIPLY(SHR(R1(I),4));
                   END;
              END;

  /* DIV */

              CALL DIVIDE;
  /* NEG */

              BRANCH$FLAG=FALSE;

  /* STP */

              CALL STOP;

  /* STI */

              CALL STORE$IMMEDIATE;


  /* RND */

              DO;
                   CALL STORE$IMMEDIATE;
                   CALL FILL(.R2,10,0);
                   R2(9)=1;
                   CALL ADD;
              END;

  /* RET */

              DO;
                   IF C$ADDR<>0 THEN
                   DO;
                        A$CTR=C$ADDR;
                        C$ADDR=0;
                        PROGRAM$COUNTER=A$CTR;
                   END;
                   ELSE CALL INC$PTR(2);
              END;
```

```
   /* CLS */

                        DO;
                             CALL SET$I$O;
                             IF WRITE$MARK THEN CALL WRITE$RECORD;
                             CALL CLOSE(C$ADDR);
                             CALL INC$PTR(2);
                        END;

   /* SER */

                        DO;
                             IF OVERFLOW  THEN PROGRAM$COUNTER = C$ADDR;
                             ELSE CALL INC$PTR(2);
                        END;
   /* BRN */

                        PROGRAM$COUNTER=C$ADDR;

   /* OPN */

                        CALL OPEN$FILE(1);

   /* OP1 */

                        CALL OPEN$FILE(2);

   /* OP2 */

                        CALL OPEN$FILE(3);

   /* RGT */

                        DO;
                             IF NOT SIGN2 THEN
                                  BRANCH$FLAG=NOT BRANCH$FLAG;
                             CALL CONDITIONAL$BRANCH(0);
                        END;

   /* RLT */

                        DO;
                             IF SIGN2 THEN
                                  BRANCH$FLAG=NOT BRANCH$FLAG;
                             CALL CONDITIONAL$BRANCH(0);
                        END;

   /* REQ */

                        DO;
                             IF NOT ZERO$RESULT THEN
                                  BRANCH$FLAG=NOT BRANCH$FLAG;
                             CALL CONDITIONAL$BRANCH(0);
                        END;

   /* INV */

                        CALL INCREMENT$OR$BRANCH(INVALID);

   /* EOR */

                        CALL INCREMENT$OR$BRANCH(END$OF$RECORD);

   /* ACC */

                        CALL ACCEPT;

   /* DIS */

                        CALL DISPLAY;

   /* STD */

                        DO;
                             CALL DISPLAY;
                             CALL STOP;
                        END;

   /* LDI */

                        DO;
                             C$ADDR(3)=CONVERT$TO$HEX(C$ADDR,C$BYTE(2));
                             CALL INC$PTR(3);
                        END;

   /* DEC */

                        DO;
                             IF C$ADDR<>0 THEN C$ADDR=C$ADDR-1;
                             IF C$ADDR=0 THEN PROGRAM$COUNTER=C$ADDR(1);
                             ELSE CALL INC$PTR(4);
                        END;

   /* STO */

                        DO;
                             CALL STORE$NUMERIC;
                             CALL INC$PTR(4);
                        END;

   /* ST1 */

                        DO;
                             CALL STORE$NUMERIC;
                             CALL SET$ZONE(C$ADDR+C$BYTE(2)-1);
                        END;
```

```
4
4      /* ST2 */
4
4                          DO;
4                              CALL STORE$NUMERIC;
5                              CALL SET$ZONE(C$ADDR);
5                          END;
4
4      /* ST3 */
4
4                          DO;
4                              CALL CHECK$DECIMAL;
5                              BASE=C$ADDR + C$BYTE(2) - 1;
5                              CALL STORE$AS$CHAR(C$BYTE(2) - 1);
5                              CALL SET$SIGN$SEP(C$ADDR + C$BYTE(2) -1);
5                          END;
4
4      /* ST4 */
4
4                          DO;
4                              CALL CHECK$DECIMAL;
5                              BASE=C$ADDR + C$BYTE(2);
5                              CALL STORE$AS$CHAR(C$BYTE(2)-1);
5                              CALL SET$SIGN$SEP(C$ADDR);
5                          END;
4
4      /* ST5 */
4
4                          DO;
4                              CALL CHECK$DECIMAL;
5                              R2(9)=R2(9) OR SIGN2;
5                              CALL MOVE(.R2 + 9 - C$BYTE(2),C$ADDR,C$BYTE(2));
5                              CALL INC$PTR(4);
5                          END;
4
4      /* LD0 */
4
4                          CALL LOAD$NUM$LIT;
4
4      /* LD1 */
4
4                          CALL LOAD$NUMERIC;
4
4      /* LD2 */
4
4                          DO;
4                              DECLARE I BYTE;
5                              HOLD=C$ADDR;
5                              IF CHECK$FOR$SIGN(CTR:=H$BYTE(I:=C$BYTE(2)-1)) THEN
5                              DO;
5                                  CALL SET$LOAD(POSITIVE);
6                                  I=I+1;
6                              END;
5                              ELSE DO;
5                                  CALL SET$LOAD(NEGITIVE);
6                                  CALL LOAD$AS$CHAR(CTR-ZONE);
6                              END;
5                              CALL LOAD$NUMBERS(C$ADDR,I);
5                          END;
4
4      /* LD3 */
4
4                          DO;
4                              HOLD=C$ADDR;
5                              IF CHECK$FOR$SIGN(H$BYTE) THEN
5                              DO;
5                                  CALL SET$LOAD(POSITIVE);
6                                  CALL LOAD$NUMBERS(C$ADDR,C$BYTE(2));
6                              END;
5                              ELSE DO;
5                                  CALL SET$LOAD(NEGITIVE);
6                                  CALL LOAD$NUMBERS(C$ADDR+1,C$BYTE(2)-1);
6                                  CALL LOAD$AS$CHAR(H$BYTE-ZONE);
6                              END;
5                          END;
4
4      /* LD4 */
4
4                          DO;
4                              HOLD=C$ADDR;
5                              IF H$BYTE(C$BYTE(2) - 1) = '+' THEN
5                                  CALL SET$LOAD(1);
5                              ELSE CALL SET$LOAD(0);
5                              CALL LOAD$NUMBERS(C$ADDR,C$BYTE(2) -1);
5                          END;
4
4      /* LD5 */
4
4                          DO;
4                              HOLD=C$ADDR;
5                              IF(H$BYTE='+') THEN CALL SET$LOAD(1);
5                              ELSE CALL SET$LOAD(0);
5                              CALL LOAD$NUMBERS(C$ADDR,C$BYTE(2)-1);
5                          END;
4
4      /* LD6 */
4
4                          DO;
4                              DECLARE I BYTE;
5                              HOLD=C$ADDR;
5                              CALL SET$LOAD(H$BYTE(I:=C$BYTE(2)-1));
5                              BASE=BASE + 9 - I;
5                              DO CTR = 0 TO I;
5                                  B$BYTE(CTR)=H$BYTE(CTR);
6                              END;
5                              B$BYTE(CTR)=B$BYTE(CTR) AND 0F0H;
5                              CALL INC$PTR(5);
5                          END;
```

```
      /* PER */

                     DO;
                          BASE=C$ADDR(1)+1;
                          B$ADDR=C$ADDR(2);
                          PROGRAM$COUNTER=C$ADDR;
                     END;

      /* CNL */

                     CALL COMP$NUM$UNSIGNED;

      /* CNS */

                     CALL COMP$NUM$SIGN;

      /* CAL */

                     CALL COMP$ALPHA;

      /* RWS */

                     DO;
                     CALL BACK$ONE$RECORD;
                          CALL WRITE$FROM$MEMORY;
                          CALL INC$PTR(6);
                     END;

      /* DLS */

                     DO;
                     CALL BACK$ONE$RECORD;
                          CALL WRITE$ZERO$RECORD;
                          CALL INC$PTR(6);
                     END;

      /* RDF */

                     DO;
                          CALL READ$TO$MEMORY;
                          CALL INC$PTR(6);
                     END;

      /* WTF */

                     DO;
                          CALL WRITE$FROM$MEMORY;
                          CALL INC$PTR(6);
                     END;

      /* RVL */

                     CALL READ$VARIABLE;

      /* WVL */

                     CALL WRITE$VARIABLE;

      /* SCR */

                     DO;
                          SUBSCRIPT(C$BYTE(2))=
                               CONVERT$TO$HEX(C$ADDR,C$BYTE(3));
                          CALL INC$PTR(4);
                     END;

      /* SGT */

                     CALL STRING$COMPARE(1);

      /* SLT */

                     CALL STRING$COMPARE(0);

      /* SEQ */

                     CALL STRING$COMPARE(2);

      /* MOV */

                     DO;
                          CALL MOVE(RES(C$ADDR(1)),RES(C$ADDR),C$ADDR(2));
                          IF C$ADDR(3)<>0 THEN CALL
                               FILL(RES(C$ADDR(1)) + C$ADDR(2),C$ADDR(3),' ');
                          CALL INC$PTR(8);
                     END;

      /* RRS */

                     DO;
                          CALL READ$TO$MEMORY;
                          CALL GET$REC$NUMBER;
                          CALL INC$PTR(9);
                     END;

      /* WRS */

                     DO;
                     CALL WRITE$FROM$MEMORY;
                          CALL GET$REC$NUMBER;
                          CALL INC$PTR(9);
                     END;
```

```
1
1        /* THIS PROGRAM TAKES THE CODE OUTPUT FROM THE COBOL COMPILER
1        AND BUILDS THE ENVIRONMENT FOR THE COBOL INTERPRETER */
1
1   100H:            /* LOAD POINT */
1
1   DECLARE
1
1   LIT              LITERALLY        'LITERALLY',
1   BOOT             LIT              '0',
1   BDOS             LIT              '5',
1   TRUE             LIT              '1',
1   FALSE            LIT              '0',
1   FOREVER          LIT              'WHILE TRUE',
1   FCB              ADDRESS          INITIAL (5CH),
1   FCB$BYTE         BASED      FCB   BYTE,
1   I                BYTE,
1   ADDR             ADDRESS          INITIAL (100H),
1   CHAR             BASED      ADDR  BYTE,
1   BUFF$END         LIT              '100H',
1   INTERP$FCB       (33)       BYTE  INITIAL(0,'CINTERP COM',0,0,0,0),
1   CODE$NOT$SET     BYTE       INITIAL (TRUE),
1   READER$LOCATION      LIT          '1C80H',
1   INTERP$ADDRESS       ADDRESS      INITIAL(2000H),
1   INTERP$CONTENT       BASED        INTERP$ADDRESS ADDRESS,
1   I$BYTE           BASED                 INTERP$ADDRESS BYTE,
1   CODE$CTR         ADDRESS,
1   C$BYTE           BASED      CODE$CTR   BYTE,
1   BASE             ADDRESS,
1   B$ADDR      -    BASED BASE       ADDRESS,
1   B$BYTE           BASED      BASE  BYTE;
1
1   MON1: PROCEDURE (F,A);
2       DECLARE F BYTE, A ADDRESS;
2       GO TO BDOS;
2   END MON1;
1
1
1   MON2: PROCEDURE (F,A) BYTE;
2       DECLARE F BYTE, A ADDRESS;
2       GO TO BDOS;
2   END MON2;
1
1
1   PRINT$CHAR: PROCEDURE(CHAR);
2       DECLARE CHAR BYTE;
2       CALL MON1(2,CHAR);
2   END PRINT$CHAR;
1
1
1   CRLF: PROCEDURE;
2       CALL PRINT$CHAR(13);
2       CALL PRINT$CHAR(10);
2   END CRLF;
1
1
1   PRINT: PROCEDURE(A);
2       DECLARE A ADDRESS;
2       CALL CRLF;
2       CALL MON1(9,A);
2   END PRINT;
1
1
1   OPEN: PROCEDURE (A) BYTE;
2       DECLARE A ADDRESS;
2       RETURN MON2(15,A);
2   END OPEN;
1
1
1   MOVE: PROCEDURE(FROM, DEST, COUNT);
2       DECLARE (FROM, DEST, COUNT) ADDRESS,
2           (F BASED FROM, D BASED DEST) BYTE;
2       DO WHILE(COUNT:=COUNT-1)<>0FFFFH;
2           D=F;
3           FROM=FROM+1;
3           DEST=DEST+1;
3       END;
2   END MOVE;
1
1
1   GET$CHAR: PROCEDURE BYTE;
2       IF (ADDR:=ADDR + 1)>=BUFF$END THEN
2       DO;
2           IF MON2(20,FCB)<>0 THEN
3           DO;
3               CALL PRINT(.'END OF INPUT   $');
4               GO TO BOOT;
4           END;
3           ADDR=80H;
3       END;
2       RETURN CHAR;
2   END GET$CHAR;
1
```

```
1
1      NEXT$CHAR: PROCEDURE;
2           CHAR=GET$CHAR;
2      END NEXT$CHAR;
1
1
1      STORE: PROCEDURE(COUNT);
2           DECLARE COUNT BYTE;
2           IF CODE$NOT$SET THEN
2           DO;
2                CALL PRINT(.'CODE ERROR$');
3                CALL NEXT$CHAR;
3                RETURN;
3           END;
2           DO I=1 TO COUNT;
2                C$BYTE=CHAR;
3                CALL NEXT$CHAR;
3                CODE$CTR=CODE$CTR+1;
3           END;
2      END STORE;
1
1
1      BACK$STUFF: PROCEDURE;
2           DECLARE (HOLD,STUFF) ADDRESS;
2           BASE=.HOLD;
2           DO I=0 TO 3;
2                B$BYTE(I)=GET$CHAR;
3           END;
2           DO FOREVER;
2                BASE=HOLD;
3                HOLD=B$ADDR;
3                B$ADDR=STUFF;
3                IF HOLD=0 THEN
3                DO;
3                     CALL NEXT$CHAR;
4                     RETURN;
4                END;
3           END;
2      END BACK$STUFF;
1
1
1      START$CODE: PROCEDURE;
2           CODE$NOT$SET=FALSE;
2           I$BYTE=GET$CHAR;
2           I$BYTE(1)=GET$CHAR;
2           CODE$CTR=INTERP$CONTENT;
2           CALL NEXT$CHAR;
2      END START$CODE;
1
1
1      GO$DEPENDING: PROCEDURE;
2           CALL STORE(1);
2           CALL STORE(SHL(CHAR,1) + 4);
2      END GO$DEPENDING;
1
1
1      INITIALIZE: PROCEDURE;
2           DECLARE (COUNT,WHERE,HOW$MANY) ADDRESS;
2           BASE=.WHERE;
2           DO I=0 TO 3;
2                B$BYTE(I)=GET$CHAR;
3           END;
2           BASE=WHERE - 1;
2           DO COUNT = 1 TO HOW$MANY;
2                B$BYTE(COUNT)=GET$CHAR;
3           END;
2           CALL NEXT$CHAR;
2      END INITIALIZE;
1
```

```
BUILD: PROCEDURE;
    DECLARE
    F2   LIT  '8',
    F3   LIT  '9',
    F4   LIT  '21',
    F5   LIT  '25',
    F6   LIT  '32',
    F7   LIT  '39',
    F9   LIT  '49',
    F10  LIT  '54',
    F11  LIT  '60',
    F13  LIT  '61',
    GDP  LIT  '62',
    INT  LIT  '63',
    BST  LIT  '64',
    TER  LIT  '65',
    SCD  LIT  '66';

    DO FOREVER;
        IF CHAR < F2 THEN CALL STORE(1);
        ELSE IF CHAR < F3 THEN CALL STORE(2);
        ELSE IF CHAR < F4 THEN CALL STORE(3);
        ELSE IF CHAR < F5 THEN CALL STORE(4);
        ELSE IF CHAR < F6 THEN CALL STORE(5);
        ELSE IF CHAR < F7 THEN CALL STORE(6);
        ELSE IF CHAR < F9 THEN CALL STORE(7);
        ELSE IF CHAR < F10 THEN CALL STORE(9);
        ELSE IF CHAR < F11 THEN CALL STORE(10);
        ELSE IF CHAR < F13 THEN CALL STORE(11);
        ELSE IF CHAR < GDP THEN CALL STORE(13);
        ELSE IF CHAR = GDP THEN CALL GO$DEPENDING;
        ELSE IF CHAR = BST THEN CALL BACK$STUFF;
        ELSE IF CHAR = INT THEN CALL INITIALIZE;
        ELSE IF CHAR = TER THEN
        DO;
            CALL PRINT(.'LOAD FINISHED$');
            RETURN;
        END;
        ELSE IF CHAR = SCD THEN CALL START$CODE;
        ELSE DO;
            IF CHAR <> OFFH THEN CALL PRINT(.'LOAD ERRORS$');
            CALL NEXT$CHAR;
        END;
    END;
END BUILD;

    /* PROGRAM EXECUTION STARTS HERE */

FCB$BYTE=0;
CALL MOVE(.('CIN',0,0,0,0),FCB + 9,7);
IF OPEN(FCB)=255 THEN
DO;
    CALL PRINT(.'FILE NOT FOUND   $');
    GO TO BOOT;
END;
CALL NEXT$CHAR;
CALL BUILD;
CALL MOVE(.INTERP$FCB,FCB,33);
IF OPEN(FCB)=255 THEN
DO;
    CALL PRINT(.'INTERPRETER NOT FOUND   $');
    GO TO BOOT;
END;
CALL MOVE(READER$LOCATION, 80H, 80H);
GO TO 80H;
EOF
```

```
        /* THIS PROGRAM TAKES THE CODE OUTPUT FROM THE COBOL COMPILER
        AND CONVERTS IT INTO A READABLE OUTPUT TO FACILITATE DEBUGGING */

100H:           /* LOAD POINT */

DECLARE

LIT             LITERALLY       'LITERALLY',
BOOT            LIT             '0',
BDOS            LIT             '5',
FCB             ADDRESS         INITIAL (5CH),
FCB$BYTE        BASED     FCB   BYTE,
I               BYTE,
ADDR            ADDRESS         INITIAL (100H),
CHAR            BASED     ADDR  BYTE,
C$ADDR          BASED ADDR           ADDRESS,
BUFF$END        LIT             '0FFH',
FILE$TYPE       DATA ('C','I','N');

MON1: PROCEDURE (F,A);
    DECLARE F BYTE, A ADDRESS;
    GO TO BDOS;
END MON1;


MON2: PROCEDURE (F,A) BYTE;
    DECLARE F BYTE, A ADDRESS;
    GO TO BDOS;
END MON2;


PRINT$CHAR: PROCEDURE(CHAR);
    DECLARE CHAR BYTE;
    CALL MON1(2,CHAR);
END PRINT$CHAR;


CRLF: PROCEDURE;
    CALL PRINT$CHAR(13);
    CALL PRINT$CHAR(10);
END CRLF;


P: PROCEDURE(ADD1);
    DECLARE ADD1 ADDRESS, C BASED ADD1 BYTE;
    CALL CRLF;
    DO I=0 TO 2;
        CALL PRINT$CHAR(C(I));
    END;
    CALL PRINT$CHAR(' ');
END P;

GET$CHAR: PROCEDURE BYTE;
    IF (ADDR:=ADDR + 1)>BUFF$END THEN
    DO;
        IF MON2(20,FCB)<>0 THEN
        DO;
            CALL P(.'END');
            CALL TIME(10);
            GO TO BOOT;
        END;
        ADDR=80H;
    END;
    RETURN CHAR;
END GET$CHAR;


D$CHAR: PROCEDURE (OUTPUT$BYTE);
    DECLARE OUTPUT$BYTE BYTE;
    IF OUTPUT$BYTE<10 THEN CALL PRINT$CHAR(OUTPUT$BYTE + 30H);
    ELSE CALL PRINT$CHAR(OUTPUT$BYTE + 37H);
END D$CHAR;


D: PROCEDURE (COUNT);
    DECLARE(COUNT,J) ADDRESS;
    DO J=1 TO COUNT;
        CALL D$CHAR(SHR(GET$CHAR,4));
        CALL D$CHAR(CHAR AND 0FH);
        CALL PRINT$CHAR(' ');
    END;
END D;


PRINT$REST: PROCEDURE;
    DECLARE
    F2   LIT   '8',
    F3   LIT   '9',
    F4   LIT   '21',
    F5   LIT   '25',
    F6   LIT   '32',
    F7   LIT   '39',
    F9   LIT   '49',
    F10  LIT   '54',
    F11  LIT   '60',
    F13  LIT   '61',
    GDP  LIT   '62',
    INT  LIT   '63',
    RST  LIT   '64',
    TER  LIT   '65',
    SCO  LIT   '66';

```

```
2          IF CHAR < F2 THEN RETURN;
2          IF CHAR < F3 THEN DO; CALL D(1); RETURN; END;
2          IF CHAR < F4 THEN DO; CALL D(2); RETURN; END;
2          IF CHAR < F5 THEN DO; CALL D(3); RETURN; END;
2          IF CHAR < F6 THEN DO; CALL D(4); RETURN; END;
2          IF CHAR < F7 THEN DO; CALL D(5); RETURN; END;
2          IF CHAR < F9 THEN DO; CALL D(6); RETURN; END;
2          IF CHAR < F10 THEN DO; CALL D(8); RETURN; END;
2          IF CHAR < F11 THEN DO; CALL D(9); RETURN; END;
2          IF CHAR < F13 THEN DO; CALL D(10); RETURN; END;
2          IF CHAR < GDP THEN DO; CALL D(12); RETURN; END;
2          IF CHAR=GDF THEN DO; CALL D(1); CALL D(SHL(CHAR,1)+5); RETURN; END;
2          IF CHAR=INT THEN DO; CALL D(3); CALL D(C$ADDR + 1); RETURN; END;
2          IF CHAR=BST THEN DO; CALL D(4); RETURN; END;
2          IF CHAR=TER THEN DO; CALL P(.'END'); GO TO BOOT; END;
2          IF CHAR=SCD THEN DO; CALL D(2); RETURN; END;
2          IF CHAR <> 0FFH THEN CALL P(.'XXX');
2      END PRINT$REST;
1
1
1          /* PROGRAM EXECUTION STARTS HERE */
1
1      FCB$BYTE=0;
1      DO I=0 TO 2;
1          FCB$BYTE(I+9)=FILE$TYPE(I);
2      END;
1
1      IF MON2(15,FCB)=255 THEN DO; CALL P(.'ZZZ'); GO TO BOOT; END;
1
1      DO WHILE 1;
1          IF GET$CHAR <= 66 THEN DO CASE CHAR;
2          ;     /* CASE 0 NOT USED */
3                CALL P(.'ADD');
3                CALL P(.'SUB');
3                CALL P(.'MUL');
3                CALL P(.'DIV');
3                CALL P(.'NEG');
3                CALL P(.'STP');
3                CALL P(.'STI');
3                CALL P(.'RND');
3                CALL P(.'RET');
3                CALL P(.'CLS');
3                CALL P(.'SER');
3                CALL P(.'BRN');
3                CALL P(.'CFN');
3                CALL P(.'CP1');
3                CALL P(.'CP2');
3                CALL P(.'FGT');
3                CALL P(.'RLT');
3                CALL P(.'RSC');
3                CALL P(.'INV');
3                CALL P(.'EOR');
3                CALL P(.'ACC');
3                CALL P(.'DIS');
3                CALL P(.'STD');
3                CALL P(.'LDI');
3                CALL P(.'DEC');
3                CALL P(.'STO');
3                CALL P(.'ST1');
3                CALL P(.'ST2');
3                CALL P(.'ST3');
3                CALL P(.'ST4');
3                CALL P(.'ST5');
3                CALL P(.'LD0');
3                CALL P(.'LD1');
3                CALL P(.'LD2');
3                CALL P(.'LD3');
3                CALL P(.'LD4');
3                CALL P(.'LD4');
3                CALL P(.'LD6');
3                CALL P(.'FER');
3                CALL P(.'CNU');
3                CALL P(.'CNS');
3                CALL P(.'CAL');
3                CALL P(.'RWS');
3                CALL P(.'CLS');
3                CALL P(.'EOF');
3                CALL P(.'WTF');
3                CALL P(.'RVL');
3                CALL P(.'WVL');
3                CALL P(.'SCR');
3                CALL P(.'SGT');
3                CALL P(.'SLT');
3                CALL P(.'SEQ');
3                CALL P(.'MOV');
3                CALL P(.'PRS');
3                CALL P(.'WRS');
3                CALL P(.'FRR');
3                CALL P(.'WRR');
3                CALL P(.'RWR');
3                CALL P(.'DLR');
3                CALL P(.'MED');
3                CALL P(.'MNE');
3                CALL P(.'GPD');
3                CALL P(.'INT');
3                CALL P(.'BST');
3                CALL P(.'TER');
3                CALL P(.'SCD');
3          END; /* OF CASE STATEMENT */
2          CALL PRINT$REST;
2      END; /* END OF DO WHILE */
1      EOF
```

```
    /* RRR */

                    DC;
                         CALL SET$RANDOM$POINTER;
                         CALL READ$TO$MEMORY;
                         CALL INC$PTR(9);
                    END;

    /* WRR */

                    CALL WRITE$RANDOM;

    /* RWR */

                    CALL WRITE$RANDOM;

    /* CLR */

                    DC;
                         CALL SET$RANDOM$POINTER;
                         CALL WRITE$ZERO$RECORD;
                         CALL INC$PTR(9);
                    END;

    /* MED */

                    DC;
                         CALL MOVE(C$ADDR(3),C$ADDR,C$ADDR(4));
                         BASE=C$ADDR(1);
                         HOLD=C$ADDR;
                         CTR=0;
                         DO WHILE (CTR<C$ADDR(1))AND(CTR<C$ADDR(4));
                              CALL CHECK$EDIT(H$BYTE);
                         END;
                         IF CTR < C$ADDR(4) THEN
                              CALL FILL(HOLD,C$ADDR(4)-CTR,' ');
                    END;

    /* MNE */

   ;

    /* GCP */

                    DC;
                         DECLARE OFFSET BYTE;
                         OFFSET=CONVERT$TO$HEX(C$ADDR(1),C$BYTE(1)-1);
                         IF OFFSET > C$BYTE + 1 THEN
                         DO;
                              CALL PRINT$ERROR('GC');
                              CALL INC$PTR(SHL(C$BYTE,1) + 6);
                         END;
                         ELSE PROGRAM$COUNTER=C$ADDR(OFFSET + 2);
                    END;

               END; /* END OF CASE STATEMENT */
          END; /* END OF DO FOREVER */
   END EXECUTE;

        /* * * * * * * * * * PROGRAM EXECUTION STARTS HERE * * * * * * */

   BASE=CODE$START;
   PROGRAM$COUNTER=B$ADDR;
   CALL EXECUTE;
   EOF
```

```
1
1      /* COBOL COMPILER - PART 2 READER */
1
1      /* THIS PROGRAM IS LOADED IN WITH THE PART 1 PROGRAM
1      AND IS CALLED WHEN PART 1 IS FINISHED.  THIS PROGRAM
1      OPENS THE PART2.COM FILE THAT CONTAINS THE CODE FOR
1      PART 2 OF THE COMPILER, AND READS IT INTO CORE. AT
1      THE END OF THE READ OPERATION, CONTROL IS PASSED TO
1      THE SECOND PART PROGRAM.                                  */
1
1
1    31COH:    /* LOAD POINT */
1
1    DECLARE
1
1    BOOT      LITERALLY '0H',
1    BDOS      LITERALLY '5H',   /* ENTRY TO THE OPERATING SYSTEM */
1    START     LITERALLY '100H',   /* STARTING LOCATION FOR PASS 2 */
1    FCB (33)  BYTE INITIAL(0,'PASS2   COM',0,0,0,0),
1    LASTDMA   ADDRESS    INITIAL(2480H), /* 80 LESS THAN MEMORY */
1    I         ADDRESS;
1
1    MONA: PROCEDURE(F,A);
2         DECLARE F BYTE, A ADDRESS;
2         GO TO BDOS;
2    END MONA;
1
1    MONB: PROCEDURE(F,A)BYTE;
2         DECLARE F BYTE, A ADDRESS;
2         GO TO BDOS;
2    END MONB;
1
1    ERROR: PROCEDURE(CODE);
2         DECLARE CODE ADDRESS;
2         CALL MONA(2,(HIGH(CODE)));
2         CALL MONA(2,(LOW(CODE)));
2         CALL TIME(10);
2         GO TO BOOT;
2    END ERROR;
1
1         /* OPEN PASS2.COM */
1    IF MONE(15,.FCB)=255 THEN CALL ERROR('02');
1         /* READ IN FILE */
1    DO I=100H TO LASTDMA BY 80H;
1         CALL MONA(26,I);   /* SET DMA */
2         IF MONB(20,.FCB)<>0 THEN CALL ERROR('R2');
2    END;
1    CALL MONA(26,80H); /* RESET DMA */
1    GO TO START;
1    EOF
```

```
1      /* COBOL COMPILER - INTERP READER */
1
1      /* THIS PROGRAM IS CALLED BY THE BUILD PROGRAM AFTER
1      CBLINT.COM HAS BEEN OPENED, AND READS THE CODE INTO MEMORY
1      */
1
1
1    80H:    /* LOAD POINT */
1
1    DECLARE
1
1    BOOT      LITERALLY '0H',
1    BDOS      LITERALLY '5H',   /* ENTRY TO THE OPERATING SYSTEM */
1    START     LITERALLY '100H',   /* STARTING LOCATION FOR PASS 2 */
1    LASTDMA   ADDRESS    INITIAL(1E80H), /* 80 LESS THAN MEMORY */
1    I         ADDRESS;
1
1    MONA: PROCEDURE(F,A);
2         DECLARE F BYTE, A ADDRESS;
2         GO TO BDOS;
2    END MONA;
1
1    MONB: PROCEDURE(F,A)BYTE;
2         DECLARE F BYTE, A ADDRESS;
2         GO TO BDOS;
2    END MONB;
1
1    DO I=100H TO LASTDMA BY 80H;
1         CALL MONA(26,I);   /* SET DMA */
2         IF MONB(20,5CH)<>0 THEN GO TO BOOT;
2    END;
1    GO TO START;
1    EOF
```

LIST OF REFERENCES

1. American National Standards Institute, COBOL Standard, ANSI X3.23-1974.

2. Aho, A. V. and S. C. Johnson, LR Parsing, Computing Surveys, Vol. 6 No. 2, June 1974.

3. Bauer, F. L. and J. Eickel, editors, Compiler Construction - An Advanced Course, Lecture notes is Computer Science, Springer-Verlag, New York 1976.

4. Digital Research, An Introduction to CP/M Features and Facilities, 1976.

5. Digital Research, CP/M Interface Guide, 1976.

6. Eubanks, Gordon E. Jr. A Microprocessor Implementation of Extended Basic, Masters Thesis, Naval Postgraduate School, December 1976.

7. Intel Corporation, 8008 and 8080 PL/M Programming Manual, 1975.

8. Intel Corporation, 8080 Simulator Software Package, 1974.

9. Knuth, Donald E. On the Translation of Languages from Left to Right, Information and Control Vol. 6, No. 6, 1965.

10. Software Development Division, ADPE Selection Office, Department of the Navy, HYPO-COBOL, April 1975.

11. Strutynski, Kathryn B. Information on the CP/M Interface Simulator, internally distributed technical note.

12. University of Toronto, Computer Systems Research Group

Technical Report CSPG-2, "An Efficient LALR Parser Generator," by W. R. Lalonge, April 1971.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Assoc Professor G. A. Kildall, Code 52Kd Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	Lt L. V. Rich, Code 52Rs Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6.	ADPE Selection Office Department of the Navy Washington, D. C. 20376	1
7.	Capt A. S. Craig, USMC 611 Canyon Drive, Springville, Utah 84663	1